IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 27, 2018, accepted February 19, 2019, date of publication March 21, 2019, date of current version April 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2905158

FTLADS: Object-Logging Based Fault-Tolerant Big Data Transfer System Using Layout Aware Data Scheduling

PREETHIKA KASU[1], TAEUK KIM[2], JUNG-HO UM[3], KYONGSEOK PARK[3], SCOTT ATCHLEY[4], AND YOUNGJAE KIM[2]

[1]Department of Computer Engineering, Ajou University, Suwon 16499, South Korea
[2]TmaxSoft, Seognam 13613, South Korea
[3]Korea Institute of Science and Technology Information, Daejeon 34141, South Korea
[4]Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (Ministry of Science and ICT) under Grant 2018R1A1A1A05079398, in part by the Korea Institute of Science and Technology (KISTI) under Grant K-17-L03-C01-S03, and in part by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, Limited Liability Company for the U.S. DOE under Contract DE-AC05-00OR22725.

ABSTRACT The layout-aware data scheduling (LADS) data movement framework optimizes congestion for end-to-end data transfers. During data transfer, LADS can avoid congested storage elements by exploiting the underlying storage layout at each endpoint. This improves the I/O bandwidth and hence the data transfer rate across high-speed networks. However, the absence of fault tolerance (FT) in LADS results in data retransmission overhead and may lead to possible data integrity issues upon faults. In this paper, we propose object-logging FT mechanisms to avoid transmitting the objects that are successfully written into the parallel file system (PFS) at the sink end. Depending on the number of log files created for the whole dataset, we have classified our FT mechanisms into three different categories: file logger, transaction logger, and universal logger. Also, to address the space overhead, we have proposed different methods of populating the log files with the information of the successfully transferred objects. We have evaluated the data transfer performance and recovery time overhead of the proposed object-logging-based FT mechanisms on the LADS data transfer framework. Our experimental results show that FT mechanisms exhibit negligible overhead ($< 1\%$) with respect to the data transfer time. However, the fault recovery time is 10% higher than the total data transfer time at any fault point.

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

I. INTRODUCTION

Large scale scientific simulations [1]–[3] and data capable Internet of Things (IoT) devices such as mobile devices, software logs, cameras, microphones, and wireless sensor networks [4] are the major sources of rapidly growing datasets. The world's technological per-capita capacity to store information has roughly doubled every 40 months since 1980s [5]. By 2025, International Data Corporation (IDC) predicts there will be 163 Zettabytes of data.

While the sheer size of the data is a major challenge, there also exist other challenges including the storage I/O bottleneck and the high data movement cost between advanced computational centers. To support an increase in the data volume, data centers are equipped with adequate storage capacity. However, at times, it is necessary to access additional data located at geographically distributed data centers. This may involve transferring huge volumes of data between the data centers. Ideally, these transfers are able to fully exploit the available network capacity between the centers.

Even though networks are reaching terabit speeds and storage capacities touching exabytes, there is a clear mismatch between network and storage speeds. This poses a major challenge in achieving higher end-to-end data transfer rates. In order to reduce the impedance mismatch between the network and storage and to improve scalability, data centers

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

deploy Parallel File Systems (PFS). Most PFS use dedicated servers to service metadata and I/O operations. In order to improve throughput, PFS 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, each with tens to hundreds of disks. Storage systems share resources between different clients. Due to this, it is possible for these clients to compete for the same resource. As contention for these resources increases, there can be a serious gap between expected and observed I/O performance [6], [7]. Also, at times, it is possible that some of the servers or their disks are overloaded while most are not. This kind of load imbalance is quite a serious problem in PFS [8]. With these observations, researchers have proposed a new bulk data transfer framework, Layout Aware Data Scheduler (LADS), which avoids temporarily congested servers during data transfers [1], [9].

One of the major challenges in the distributed environment is fault handling; hardware, network, and software 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.

LADS exploits the underlying storage layout at the source and sink to maximize throughput without negatively impacting the performance of shared storage resources for other users. LADS focuses on objects, rather than files, which allows the LADS framework to implement layout-aware scheduling algorithms. Due to this object layout-aware scheduling, objects might be transferred out-of-order from source to sink. If any object is lost due to any of the faults along the end-to-end path, this will result in data integrity and performance issues. The current LADS framework, however, does not offer any solution [10] to the faults occurring in the end-to-end path. This results in retransmitting the whole file (or objects) upon fault, causing unnecessary congestion [11], [12].

Due to this *out-of-order* nature of object transmission, checkpoint-based logging of file offsets [13] or logging the index of last transferred object is not enough for resuming the transfers upon fault. Another approach is to maintain the log of all objects that were successfully sent and written at the sink end PFS. This kind of logging mechanism [14] will have an impact on the overall space occupied by the log file. Also, the amount of time consumed for logging successful objects while transferring and for retrieving the successfully completed objects upon fault will have a direct impact on the overall performance of the data transfer. Our main objective is to design an object logging-based FT mechanism to minimize the time, space and retrieval overhead while not negatively impacting the performance of the data transfers.

In this paper, we propose an object logging-based FT mechanism, to use in conjunction with LADS. In order to analyze the performance and space overhead of the FT mechanisms on LADS, we propose different FT mechanisms. This paper makes the following contributions.

- In object based logging, each and every file in the dataset is associated with one log file. Due to this, as the dataset size increases, the number of log files also increases. This causes non-negligible overhead on the file system. When a new file is created, the system-wide open file table in the kernel as well as per-process file table will be updated. When threads concurrently try to update the shared table, it leads to contention. To avoid this, we implement a light-weight logging mechanism.
- Depending on the number of log files generated per dataset, we proposed three different object based FT mechanisms: *File Logger*, *Transaction Logger*, and *Universal Logger* [15]. With the File Logger mechanism, each file in the target dataset is associated with one log file, which will be used for recovery upon resuming from the fault. For both the Transaction and Universal Logger mechanisms, one log file is associated with one transaction and whole dataset respectively.
- The space overhead depends on how the completed object information is populated in the log files. To optimize space, we have proposed different logging methods: *char*, *int*, *enc*, *binary*, *bit8* and *bit64* [15]. We have evaluated space overhead using these logging methods with the above mentioned logger mechanisms.
- We have analyzed the overhead of an object based FT mechanism(s)/method(s) with respect to data transfer performance and space overhead. For evaluating our implementation, we have used a Luster 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 60 KB (KiloBytes). However, total recovery time is 10% higher than that of the total data transfer time at any fault point.

The rest of the paper is organized as the following: Section II describes LADS background followed by the motivation of our work. Section III reviews the LADS system architectural details. Section IV presents the proposed object-based logging mechanisms to support FT with the LADS framework. Section V describes the design and architectural changes incorporated in LADS to support FT. The experimental results and related works are presented in Section VI and Section VII. We conclude the paper in Section VIII.

II. BACKGROUND AND MOTIVATION

A. LAYOUT-AWARE DATA SCHEDULING

High speed networks and relatively slower storage servers work together, and sometimes against each other, while transferring data between data centers [1], [9]. A storage server might experience congestion if the number of I/O requests exceed the storage server capabilities. Due to this congestion, the storage server consumes more time to service new I/O requests. This kind of behavior is common and is expected within a PFS when multiple applications or a single large

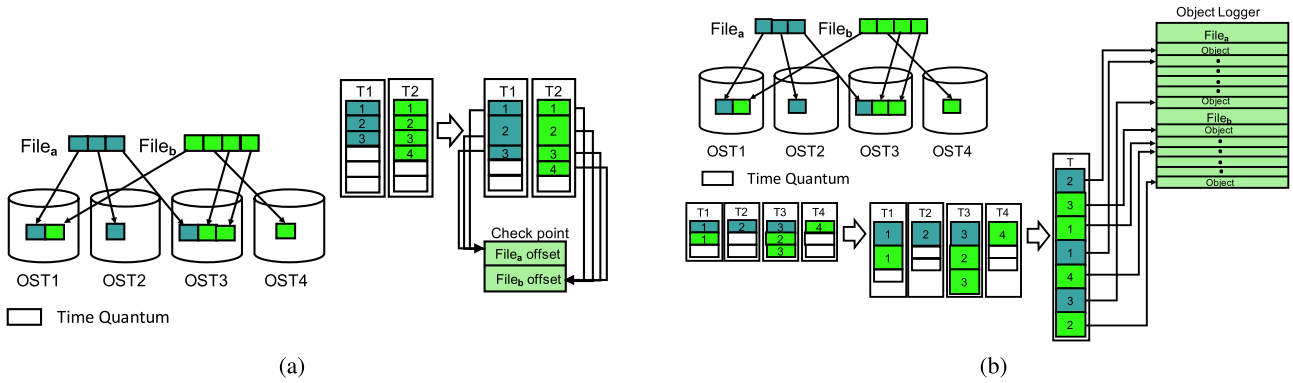


FIGURE 1. File based and object based logging.

application is trying to access files on the same object storage target. To some extent, it is possible to avoid this kind of congestion issues by using OS caching and application level buffering techniques. But big data transfer tools [16], [17] can not benefit from these kinds of techniques due to the high volume of the data. If the source end is congested during a data transfer due to large read requests, the source end will not be able to load the data into the network buffers at the expected rate. This will cause network buffers to drain and stall the transfer. On the other hand, if the sink end is congested, due to large write operations, the sink end will not be able to consume the data at the expected rate. This will consume the sink buffers and eventually stall the I/O threads due to unprocessed buffers.

Existing big data transfer tools [16], [18] consider the workload in terms of logical files and they do not consider the physical distribution of files in the PFS. Due to this, if one I/O thread is assigned to transfer a file, it will read or write the file sequentially until the entire file is transferred. If part of the file resides on a congested server, the file transfer stalls. To improve the data transfer performance, it is possible to assign multiple I/O threads to process the data transfer. Employing multiple I/O threads without the knowledge of the physical distribution of the file might still result in disk contention issues as multiple threads compete for the same OSS (Object Storage Servers) or OST (Object Storage Target).[1] Because of this contention, data transfer performance of the application will degrade.

The LADS [1], [9] data transfer framework addresses storage contention issues by considering the physical distribution of the file over different OSTs. LADS considers the workload as collection of objects rather than files. The workload is divided into O objects, where O is the objects of N total files, and each object represents one *MTU* (Maximum Transmission Unit) of data. LADS avoids the OST contention by scheduling accesses of OSTs. Due to this, objects of different logical files might be transferred in parallel. Also, LADS framework is implemented using Common Communication Interface (CCI) [19], [20], which utilizes zero-copy and OS bypass hardware features, when available, to improve the transfer rate.

In a PFS, each file is striped over multiple OSTs to improve the overall I/O throughput. LADS improves the data transfer performance by exploiting the PFS layout. As the file is distributed over N OSTs, LADS employs N threads to request N objects each from separate OST. If any of the requests is delayed by a congested OST, the N-1 threads are free to issue new requests to other OSTs. By the time a request to the congested server completes, other threads of LADS will be able to retrieve more than N objects. With this, the overall data transfer parallelism and the data throughput improves. Although the LADS framework exhibits higher throughput than existing tools, the lack of FT support to handle software, hardware or network faults during the transfer necessitates that a new instance of LADS will retransmit all the objects of whole dataset upon recovery from the fault.

B. MOTIVATION

Traditional big data transfer tools, like bbcp, rely on the logical view of the files, which ignores the underlying storage architecture. Due to this logical nature of the transfer, objects of the same file are transferred in sequence. As shown in Figure 1(a), even though there is a possibility of resource contention between threads, T_1 and T_2, all the objects of $File_a$ and $File_b$ are transferred in sequence. Thread T_1 transfers the first object of the $File_a$ and then records file offset information. After completing the second object, the threads overwrites the checkpoint record with the updated file offset information. This process will be continued for all the files in the dataset. During this process, if the transfer is resumed from fault, the transfer tool checks if checkpoint record exists for the target file or not. If it exists, it will start transferring the objects beginning from the offset found in the checkpoint record.

In contrast to the traditional big data transfer tools, LADS exploits underlying storage architecture and views the files in the physical point of view. This way it is possible to

[1] We use Luster terminology for OSS and OST. An OST manages a single device. A single Luster OSS manages one or more OSTs.

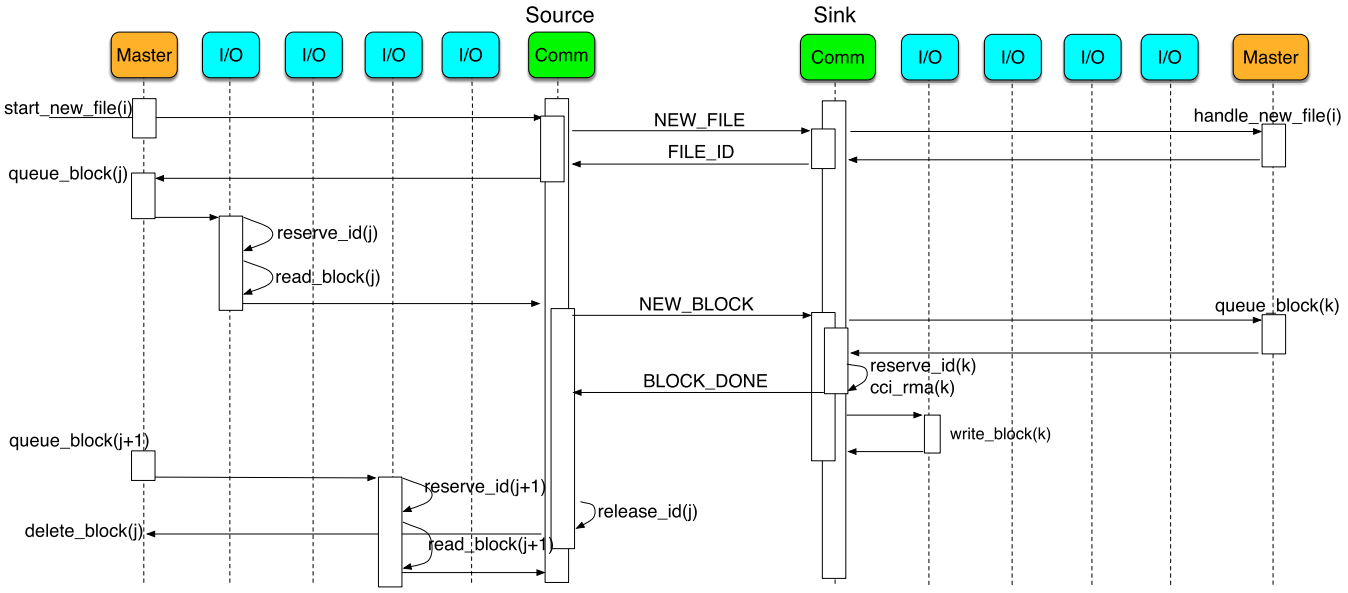


FIGURE 2. LADS data transfer sequence diagram.

avoid unwanted OST resource contention. LADS considers the entire workload as O objects, where O is the total objects in the N total files, and each object represents one transfer MTU of data. As shown in Figure 1(b), a thread can be assigned to an object of any file on any OST without requiring all objects of a particular file be transferred before objects of another file. From the Figure 1(b), we can observe that the second object of $File_a$ is transferred first before the first object is transferred. Similarly, we can also observe the out-of-order object transfer for $File_b$. This process repeats for all the files in the dataset. As objects are transferred out of order, it is not possible to recover the completed object information by a checkpoint logging based file offset as shown in Figure 1(a). Instead, we need to devise a mechanism in which we can retrieve all the completed objects that are successfully transferred prior to the fault. To achieve this, one method is to maintain the information about all the objects of all the logical files that are successfully transferred.

The major issue is the amount of space occupied by the log files in case of big dataset and also the logging overhead on the data transfer rate. So our work is motivated to answer the following questions:

- How to minimize the object based FT overhead on the LADS data transfer rate?
- How to minimize the number of files created while processing data transfer?
- How to minimize the space occupied by the object based FT method?
- How to reduce the recovery time while resuming the transfer?

To address the above challenges, we have proposed object based FT mechanisms to be used in conjunction with LADS tool.

III. LADS ARCHITECTURE

In this section, we first describe the LADS framework implementation details. Next, we discuss the possible performance issues with the LADS framework when used in faulty environments.

A. LADS OVERVIEW

LADS [1], [9] system is implemented by having one *master* thread, a configurable number of *I/O* threads and one communication(*comm*) thread. The *master* thread is responsible for scheduling object transfers and the *I/O* threads read or write the object data from or to the PFS. The *comm* thread handles the communication between the source and sink servers. The *master* and *I/O* threads block while waiting for a resource, however, the *comm* thread always progresses the communication between source and sink.

When initiating a transfer, the source and sink processes (hereafter simply Source and Sink) initialize the threads necessary for the communication along with all the required locks, wait queues, OST work queues and allocate RMA buffers used for data transfer. The *comm* thread, which communicates using CCI, opens a CCI endpoint and registers a RMA buffer with CCI. The Sink *comm* thread opens a CCI endpoint and waits for the connection from the Source. The Source *comm* thread establishes connection with the open Sink CCI endpoint. During connect request, the Source *comm* thread, sends its maximum object size (set to 1MB in our evaluation), the number of objects in the RMA buffer, and the memory handle for the RMA buffer. The Sink *comm* thread accepts the connection request, which triggers the CCI connect event on the Source.

The sequence of a data transfer between Source and Sink endpoints is shown in Figure 2. For each file in the

target dataset, the Source *master* thread generates a NEW_FILE request and enqueues the same with the work queue of the *comm* thread. The Source *comm* thread dequeues the request and transfers the same to Sink. The Sink *comm* thread receives a NEW_FILE request and enqueues the same to *master* thread's work queue and wakes it up. Based on the target file information in the request, the *master* thread opens the file and adds the file descriptor to the FILE_ID request and then enqueues the same on *comm* thread's work queue. The Sink *comm* thread dequeues the request and sends it to Source. On receiving the FILE_ID request, the Source *comm* thread enqueues the request on *master* thread's wait queue and wakes it up. The Source *master* thread splits the file into object sized blocks and generates NEW_BLOCK requests and enqueues the requests on *I/O* thread's wait queue and wakes it up. Based on the first NEW_BLOCK request, *I/O* thread determines the OST to be used for reading the object data and issues pread() to read the object data into the RMA buffer registered with CCI. After completing the read operation, it enqueues the request on the *comm* thread's work queue. The *comm* thread dequeues the request and transfers it to the Sink.

The Sink *comm* thread receives the NEW_BLOCK request and then tries to reserve a RMA buffer. If a RMA buffer is available, it then initiates a RMA read operation. If it fails to get a RMA buffer, it enqueues the request on *master* thread's work queue. The *master* thread waits until the RMA buffer is available. Once the buffer is available, *master* thread enqueues the request on *comm* thread's queue, which issues the RMA read operation. Upon successful completion of the RMA read, the Sink *comm* thread sends a BLOCK_DONE request to the Source and wakes up an *I/O* thread. The *I/O* thread dequeues the request and calls pwrite() to write the data to disk. On completing the write operation, the *I/O* thread releases the RMA buffer, so the *comm* thread can initiate another RMA Read. This process is repeated until all the objects of the dataset are successfully transferred to Sink.

B. PROBLEM DEFINITION

Fault Tolerance: Often during large transfers, the connection between the transferring systems is lost. Connection errors might be the result of software, hardware or network faults. The data transfer tool ability to resume the transfer from where it let off avoids transferring already completed files or objects. This not only avoids congestion due to redundant file transfer but also improves the overall transfer performance in faulty network conditions. The LADS framework ensures successful transmission of the objects to the Sink. But, if there is an error while writing to PFS, it will go unnoticed and the transferred data will not be useful for further analysis due to data corruption. Also, if there is an error during the transfer, the LADS object transfer protocol restarts the transfer from the beginning. This not only wastes the resources but also increases the overall transfer time. We propose solutions to avoid transferring redundant data by implementing a light-weight object-based logging FT mechanisms with the LADS framework.

IV. OBJECT BASED LOGGING

As LADS employs layout-aware and OST congestion aware I/O scheduling algorithms, objects of any file on any OST can be transferred before objects of another file or of the same file. Because of this, it is possible that objects of the same logical file might be transferred out of order. Due to this *out-of-order* nature of the transmission, logging file offset based FT mechanisms can not be employed with LADS. To support FT with LADS, information of all the objects of a logical file that are successfully written to PFS at the Sink need to be maintained. This process is not only computationally expensive but also results in additional space overhead. In order to minimize the computational and space overhead, we proposed different logging mechanisms.

In this section, we describe the proposed object-based logging mechanisms [15] to support FT with LADS.

A. OBJECT BASED LOGGING MECHANISMS

Depending on the number of log files generated per dataset, we propose three different object based FT mechanisms: File Logger, Transaction Logger, Universal Logger.

1) FILE LOGGER

Using the File Logger object logging mechanism, one log file is created corresponding to each file being transferred. For example, consider $File_A$ needs to be transferred to another data center for analysis. LADS data transfer tool segments the file into N objects or blocks. Upon successful completion of block K (B_k), the File Logger mechanism will write to the log file that B_k was successfully written to the PFS at Sink. Due to any fault, if it needs to restart the transfer, LADS first searches for the completed blocks from the corresponding log file and builds the list of the blocks that still need to be transferred. Once all the blocks, corresponding to one logical file has been successfully transferred and written to the PFS, the log file will be deleted.

This mechanism is easy to implement. Also, as the log is maintained for a single file, the search overhead for retrieving the completed block information is minimal. However, as each and every file is associated with one log file, an increase in the number of files in the dataset will have a direct impact on the number of log files created. To avoid this, we implement a light-weight logging mechanism. Using light-weight logging, log files are created only when the first object of the new file is transferred successfully and deleted upon completion of the transfer.

2) TRANSACTION LOGGER

In contrast to the File Logger mechanism, the Transaction Logger makes use of one log file for one transaction which may be a set of files. The size of the transaction can be configurable depending on the total dataset size.

Using a single log file for maintaining the completed objects' information of multiple logical files requires methods to differentiate the objects of one file from another. To achieve this, an index file is used. The index file contains the information of the file being transferred. Each line in the index file looks like,

[LogFileName, FileName, TotalBlocks, Offset, Data_Length]

where,

LogFileName → Transaction Logger file name
FileName → Name of the file being transferred
TotalBlocks → Number of Blocks
Offset → Offset in the log file
Data_Length → Length of data in log file

With this logger mechanism, the search overhead for retrieving the completed block information is similar to File Logger mechanism. However, computational complexity is higher than file logger mechanism due to the presence of multiple files objects rather than single file objects.

3) UNIVERSAL LOGGER

The Universal Logger mechanism is similar to the Transaction Logger in the way the completed block information is logged. In contrast to the Transaction Logger, the Universal Logger makes use of a single log file corresponding to all the files in the dataset. The single log file is used for maintaining the completed object information of multiple logical files. This needs a method to differentiate the objects of one file from another. To achieve this, an index file is used. The index file contains the information of the file being transferred. Each line in the index file looks like,

[FileName, TotalBlocks, Offset, Data_Length]

where,

FileName → Name of the file being transferred
TotalBlocks → Number of Blocks
Offset → Offset in the log file
Data_Length → Length of data in log file

With the Universal Logger mechanism, the search overhead and computational complexity for logging the completed blocks information is similar to the Transaction Logger.

B. OBJECT BASED LOGGING METHODS

To optimize the space overhead, the logger mechanisms described above are analyzed with different object based logging methods. These methods vary on how log information is stored.

- Char type: The block number to be populated in the log file will be converted to string first and then written to the file.
- Encoding type: Successful block information with the char type will be encoded using a Variable Length Datatype (VLD) library written by one of the authors.
- Int type: Successful block will be written to the file using integer data.
- Binary type: Before writing to the file, each block number is first converted to binary format. Assuming any file under consideration is not segmented more than 2^{32} number of blocks, currently we are using 32-bit binary representation.
- Bit binary: Each bit is used to represent one block. For example, transferring block K has been completed successfully and considering N-bit approach, we can represent completed block in this method by calculating the array index (i) and bit position (j) as, $Array_i = K / N$ and $Bit_j = K \bmod N$. Setting the bit in the $Bit_{Position}$ of the corresponding index, $Array_{index}$ will indicate the completion of the transfer of that particular block. In this method, we compare the space and execution time by using both 8-bit and 64-bit. Pseudo code for the bit-binary method of logging is as shown in Algorithm 1.

Algorithm 1 Bit Binary Method of Logging (N = 8 or N = 64)

```
1: procedure BITBINARY(A, N)
2:     buff < − ReadFromFile;
3:     ArrayIndex = A/N;
4:     Bit_Pos = A%N;
5:     buff[ArrayIndex] = buff[ArrayIndex] |
6:         (1 << Bit_Pos)
7:     WritetoFile < − buff;
8: end procedure
```

V. FAULT-TOLERANCE DESIGN WITH LADS

Fault Tolerance support for the Layout-Aware Data Scheduling (FT-LADS) framework is motivated to answer a simple question: how can we improve LADS performance in case of software, hardware or common communication errors?

In this section, we describe the design and architectural changes incorporated in LADS to support FT.

A. FT LOG FILES

The LADS data transfer is driven by the Source. Upon initiating the transfer, the Source *master* thread generates the list of objects to be transferred to the Sink. If the Source has prior knowledge of the completed objects, then it is possible to exclude those objects from the list. All our proposed object logging mechanisms generate FT log files at the Source.

B. SEQUENCE FLOW OF FT-LADS

The proposed FT-LADS communication protocol between the Source and the Sink is as shown in Figure 3. The BLOCK_DONE message in LADS has been modified to a BLOCK_SYNC message to handle data transmission errors as well as PFS write failures as listed in Listing 1. Upon receiving the BLOCK_SYNC message, based on the synchronous or asynchronous logging method, the Source *comm* thread either writes the completed block information to the

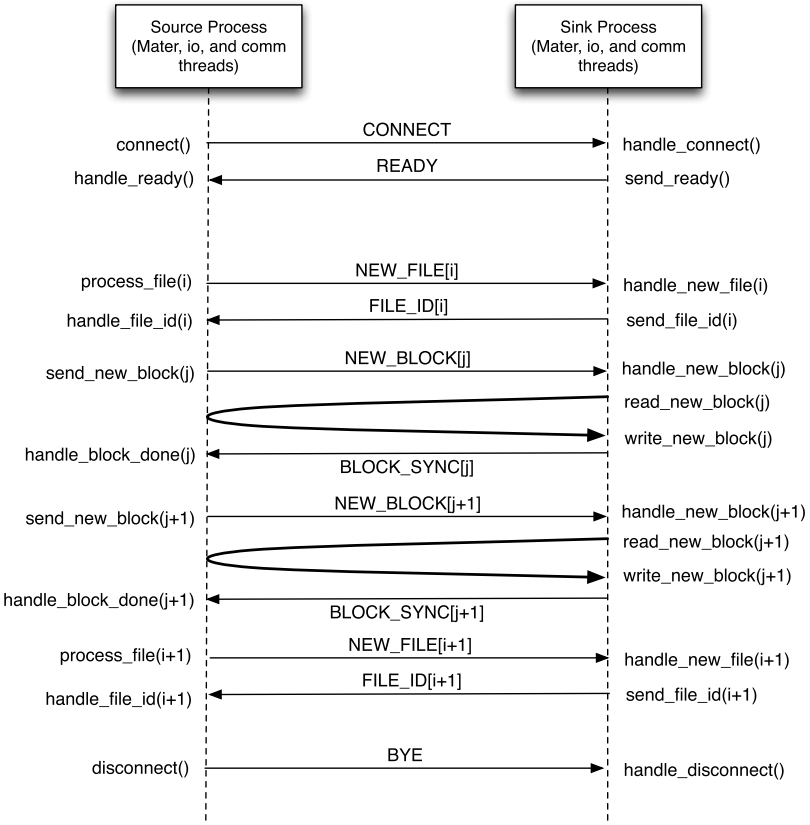


FIGURE 3. Communication protocol between source and sink.

```
typedef enum msg_type {
    CONNECT = 0,     //Connect Request
    NEW_FILE,        //New File request
    FILE_ID,         //Sink File ID.
    NEW_BLOCK,       //Ready for RMA Read
    BLOCK_SYNC,      //Sync with Sink PFS
    BYE,             //ready to disconnect
    FILE_CLOSE,      //file close
} msg_type_t;
```

Listing 1. Communication message type.

FT log file directly or enqueues the request on the wait queue in the *logger* thread. In case of synchronous logging, the completed objects information is populated to the FT log file in the context of the *comm* thread. In the case of asynchronous logging, a different *logger* thread is used for logging the completed objects information to the log file. In both cases, we implemented and evaluated the performance and found no difference between the two methods. Therefore, we present only the synchronous logging mechanism.

The data flow in FT-LADS is shown in the Figure 4. For each file in the target dataset, the Source *master* thread generates a NEW_FILE request and enqueues the request on the work queue in the *comm* thread. The *comm* thread dequeues the request and transfers it to Sink. The Sink *comm* thread receives the NEW_FILE request and enqueues the request to the *master* thread's work queue and wakes it up. Based on the target file information in the request, the *master* thread opens the file and adds the file descriptor to the FILE_ID request and then enqueues the request to the *comm* thread's work queue. The *comm* thread dequeues the request and sends it to Source. On receiving the FILE_ID request, the Source *comm* thread enqueues the request on the *master* thread's wait queue and wakes it up. The *master* thread splits the file into object sized blocks and generates the NEW_BLOCK requests and enqueues the requests on the *I/O* thread's wait queue and wakes it up. An *I/O* thread first reserves a buffer registered with CCI for RMA. It then determines which OST queue it should access and then dequeues the first NEW_BLOCK request. It uses pread() to read the data into the RMA buffer. When the read completes, it enqueues the request on the *comm* thread's work queue. The *comm* thread dequeues the request and transfers it to the Sink. At the Sink, the *comm* thread receives the request and attempts to reserve a RMA buffer. If successful, it initiates an RMA read operation of the data. If not, it enqueues the request on the *master* thread's work queue and wakes the *master* thread. The *master* thread will sleep on the RMA buffer's wait queue until a buffer is released. Once the buffer is available, the request is placed on the *comm* thread's queue, which will issue RMA read operation. Upon completing the RMA read operation, the Sink's *comm* thread determines the appropriate OST by

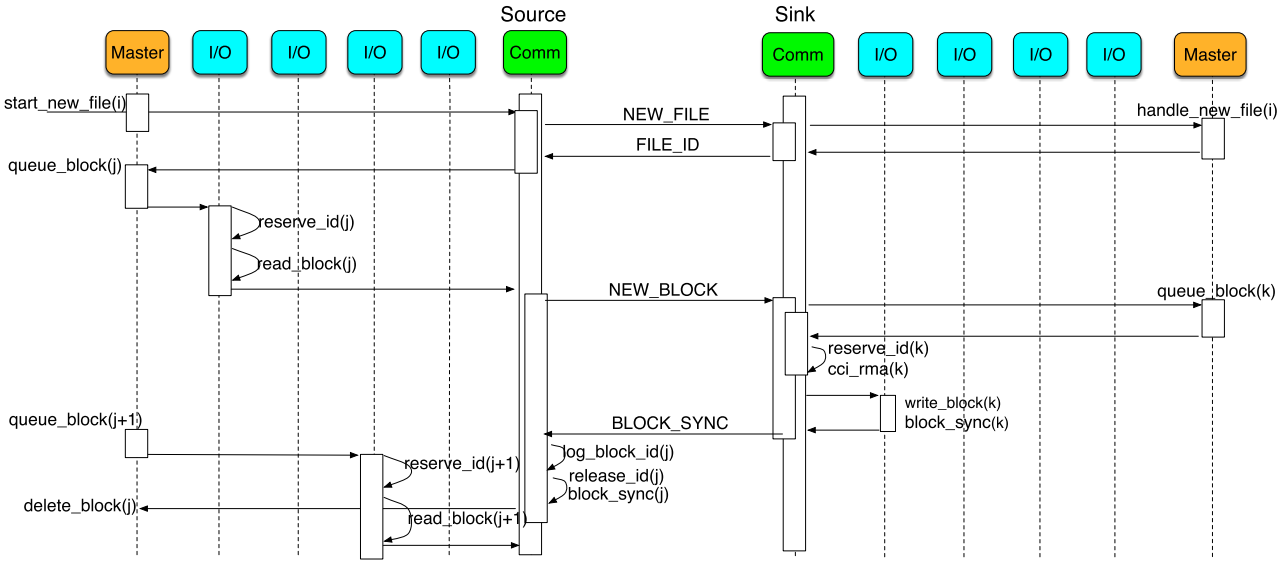


FIGURE 4. Fault tolerant LADS data transfer sequence diagram.

the object's file offset and queues it on the OST's work queue. It then wakes up an *I/O* thread. The *I/O* thread looks for the next OST to service and dequeues a request and then calls pwrite() to write the data to the disk. When the write is completed, it releases the RMA buffer so that the *comm* thread can initiate another RMA read operation and also sends the BLOCK_SYNC request to Source. Upon receiving the BLOCK_SYNC message, the Source *comm* thread, based on synchronous or asynchronous logging method, logs the completed block information to the FT file or enqueues the request to the *logger* thread wait queue respectively. The logging method will vary based on the logger mechanism. This process is continued until all the objects of the dataset are successfully transferred to the Sink or until any interruption due to fault.

C. RESUMING FAILED TRANSFERS

A fault tolerant design enables LADS to resume with the current data transfer from the same point as it was interrupted, upon recovery. When a transfer is initiated, based on the selected object logger mechanism and method as described in Section IV, a log file will be created in *ftlads* subdirectory under the user's home directory. If a data transfer is initiated by enabling the FT option, this subdirectory will be created automatically. The actual log file name under this subdirectory will vary based on the selected object logger mechanism.

This section describes the steps performed before and after a fault in order to resume the transfers.

1) BEFORE FAULT

Upon scheduling the data transfer, the Source creates a NEW_FILE request with the current file's metadata and sends the request to Sink. Based on the NEW_FILE request information, the Sink opens a file, creates a FILE_ID request with Sink end file descriptor and sends it to Source. On receiving the FILE_ID request, the Source schedules all the objects of the file for transfer. Upon successful transfer and writing to the PFS, a BLOCK_SYNC message will be sent from Sink to Source. On receiving the BLOCK_SYNC message, Source writes the completed object information to the FT log file. If all the objects are successfully transferred, then the FT log entry corresponding to that file is deleted.

2) AFTER FAULT

On resuming the transfer, the Source creates a NEW_FILE request with the current file's metadata and sends the request to Sink. On receiving the NEW_FILE request, Sink checks if the file already exists and the file's metadata is matching with the Source file's metadata. If matching, the file is marked as completed and will be excluded from the files to be transferred list. If the file does not exist or the metadata does not match, the Sink creates a FILE_ID request and sends it to Source. Upon receiving the FILE_ID request, Source checks if the FT log file corresponding to the file exists in the FT logger directory. If it exists, the objects that were successfully transferred are retrieved. Then, Source builds the object list by excluding already completed objects and then schedules the transfer.

VI. EVALUATION

For the evaluation of FT-LADS, we have created a simulation environment where we have induced hardware faults during data transfer. First, we evaluate the FT overhead on LADS by showing the results of FT-LADS without fault. Then we explore the effectiveness of FT-LADS by comparing the recovery time overhead in FT-LADS with bbcp by varying fault points. All our experiments were conducted under similar conditions.

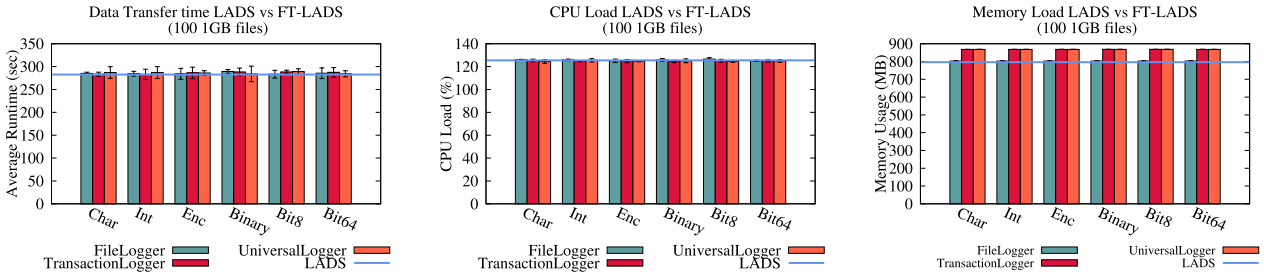


FIGURE 5. Performance comparison of LADS and FT-LADS for big workloads. The 99% confidence intervals are shown in error bar. (a) Data transfer time. (b) CPU load. (c) Memory usage.

A. EXPERIMENTAL ENVIRONMENT

1) IMPLEMENTATION

FT-LADS, which is based on a server-client model, has been implemented using $6K$ lines (including both LADS and FT implementation) of C code using pthreads.

2) TEST-BED

For our experiments, we used a private testbed with two nodes (Source and Sink) connected by InfiniBand (IB). The nodes use the IB network to communicate with each other. We have used Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz servers with 24 cores and 128 GB DRAM. Both Source and Sink hosts are running with Linux kernel 3.10.0-514.21.1. Also, the Source and the Sink nodes have separate Luster file systems 2.9.0 [21] with one OSS and 11 OSTs, mounted over 1 TB drives each. By default, our Luster file system configuration includes stripe count of one with stripe size of 1 MB. To fairly evaluate our implementation, we have ensured that the storage server bandwidth is not over-provisioned with respect to the network bandwidth between those Source and Sink servers (i.e., the network would not be the bottleneck).

3) WORKLOADS

It is observed that 90.35 percent of the files are less than 4 MB and 86.76 percent are less than 1 MB [1], [9]. Less than 10 percent of the files are greater than 4 MB whereas the larger files occupy most of the file system space. For the purpose of evaluation, we had used two groups of files with different sizes; one for small workloads with 10,000 1 MB files, and the other for big workloads with 100 1 GB files. For evaluation, we have pre-populated Source's file system with big and small workloads where each file's stripe count is 1 and size is 1MB.

4) CONFIGURATION

Experimental results presented in LADS [1], [9], suggest that LADS data transfer performance increases linearly with the number of I/O threads. To have an optimal evaluation environment, in all our experiments, we have configured FT-LADS to use 4 I/O threads, 1 master thread, and 1 comm thread.

With the Transaction Logger, we have considered 4 files in one transaction. If the transaction size is set to 1, then the Transaction Logger is same as the File Logger mechanism, as each and every file will be associated with one log file. If the transaction size is set to maximum, then the Transaction Logger is same as the Universal Logger. So for our evaluations, we have used intermediate size as transaction size.

All the experiments were done by utilizing a large, fixed amount of DRAM used as RMA buffers at both the Source and the Sink. Our current implementation makes use of max 256 MB of DRAM at both Source and Sink. We have run multiple iterations of all the experiments and shown average as bar graph. Also, 99% confidence intervals are shown in error bar, wherever needed.

5) RECOVERY TIME

As there is no direct method of evaluating the recovery time, we have estimated the recovery time of failed transfers as below.

$$ER_t = TBF_t + TAF_t - TT_t$$

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 (1)

B. PERFORMANCE COMPARISON WITH LADS

One of the major objectives while designing the object based FT mechanisms is to minimize the object based FT overhead on LADS data transfer time. In this section, we present the evaluation results of different object based FT mechanisms and methods described earlier (Section IV). For evaluating the data transfer rate and computational overhead of FT-LADS, we have used total time to transfer, CPU load and memory usage as performance factors.

Figure 5 and 6 shows the performance comparison between LADS and FT-LADS. In these figures, the proposed object based mechanisms are represented using bar graphs, in which a line is used to represent LADS.

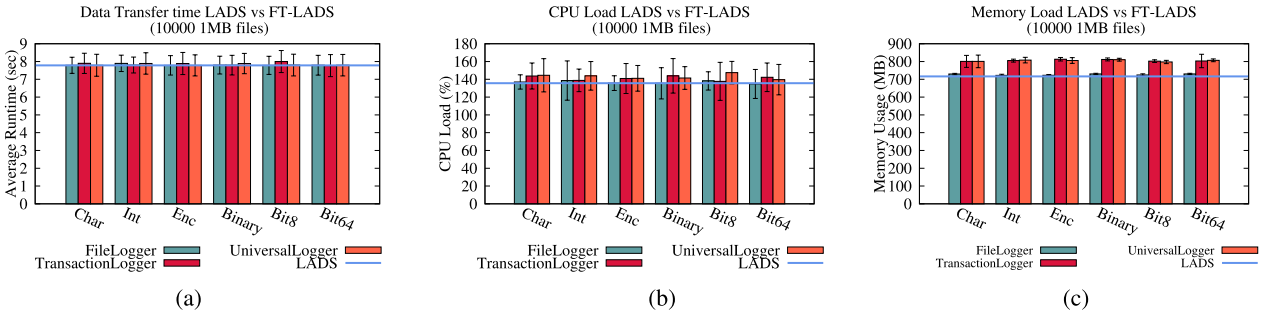


FIGURE 6. Performance comparison of LADS and FT-LADS for small workloads. The 99% confidence intervals are shown in error bar. (a) Data transfer time. (b) CPU load. (c) Memory usage.

Figure 5(a) and Figure 6(a) depicts the total time consumed for transferring the big workloads and the small workloads respectively. From Figure 5(a) and Figure 6(a), we can clearly observe that all the proposed FT mechanisms have negligible impact on the overall data transfer time. Therefore, we can conclude that the proposed FT mechanisms have no impact on the overall data transfer rate, as data transfer time is inversely proportional to the data transfer rate.

Total CPU load during data transfer is another important design aspect while designing FT support with LADS. Figure 5(b) and Figure 6(b) depict the CPU load while processing the data transfer. From these figures, we can observe that there is no significant impact on the total CPU load with FT support, compared with LADS.

Figure 5(c) and Figure 6(c) represent memory load comparison of proposed FT mechanisms with LADS. From these figures, we can clearly observe that, with the File Logger mechanism, there is no impact on the memory load, whereas, with other mechanisms, we can see an increase in the memory load. In the case of the File Logger mechanism, we simply write the completed object information to the corresponding FT logger file and there are no additional data structures which will be used to save the intermediate data. In the cases of the Transaction and Universal Logger mechanisms, the completed object information of multiple files are interleaved into a single log file. This increases the recovery time upon fault. To optimize the recovery time, the completed object information of all files are maintained internally as a list before actually logging into the log file. Due to the use of an intermediate data structure, the total memory used by Transaction and Universal mechanisms is higher than those of File Logger and LADS.

From the 99% confidence intervals which are shown as error bars in Figure 5 and 6, we can observe that there is a lot of variability for small workloads with respect to data transfer time, CPU load and memory load. This variability might be due to the file management overhead of the file system, as the number of files to be transferred is much higher in small workloads.

As shown in Figure 5 and 6, the performance is not affected by the FT methods (Char, Int, Enc, Binary, Bit8 and Bit64) used for both big and small workloads. With this, we can conclude that all the proposed object based FT mechanisms and methods have a negligible performance overhead compared to LADS and the File Logger is the most lightweight mechanism with minimal overhead among the proposed FT mechanisms.

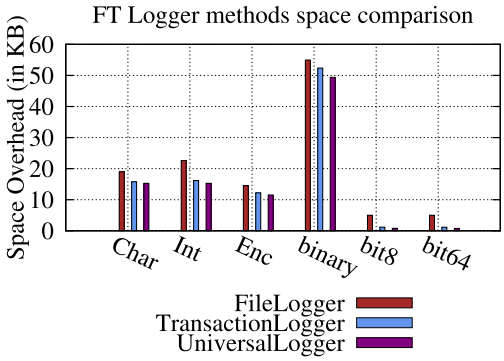


FIGURE 7. FT Logger methods space overhead.

C. OBJECT BASED LOGGER METHODS SPACE ANALYSIS

Another important aspect while designing the FT-LADS framework is the amount of space occupied by the log files during data transfer. To optimize the log space occupied, as mentioned in Section IV-B, we have proposed three different logging methods. In this section, we compare the space occupied by the different logger methods.

Figure 7 depicts the space overhead of all the proposed logging methods for all the object based FT mechanisms. From the figure, it is evident that bitbinary (Bit8 and Bit64) method is the most effective among all the logger methods due to its low space overhead. This is expected as each object is represented with one bit. Though other logging methods have relatively higher space overhead than bitbinary method, the overhead is quite negligible which is in the order of few KB.

As mentioned in section VI-B, all the proposed FT methods have negligible performance overhead among each other. With this, we can conclude that Bit8 and Bit64 FT methods are recommended with respect to space overhead with the proposed object based FT mechanisms.

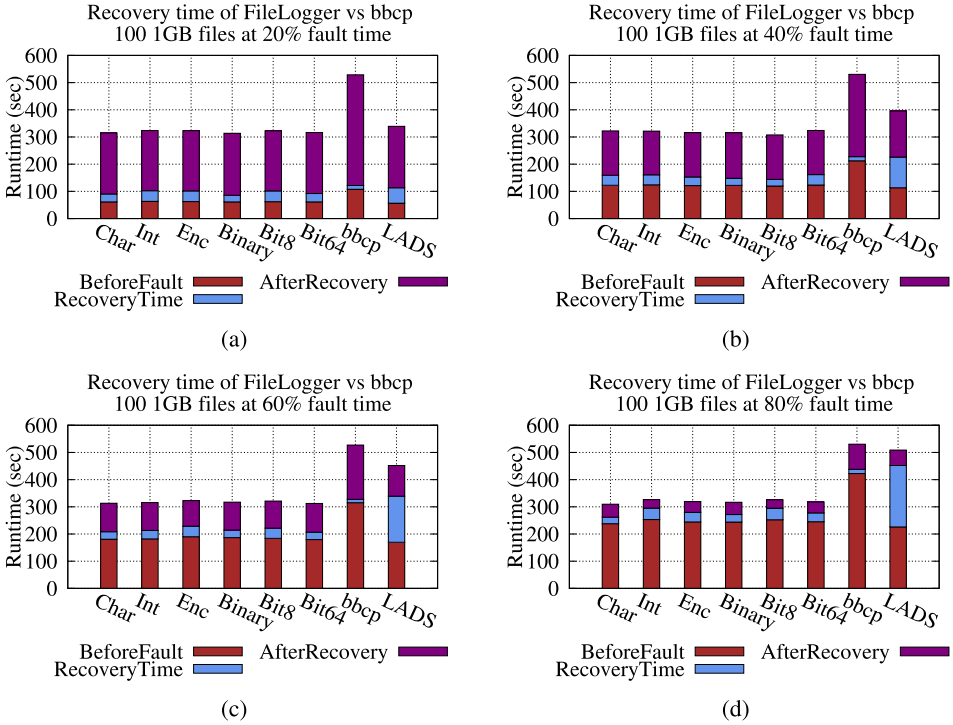


FIGURE 8. Recovery time analysis of FileLogger at varying fault timing for big workloads. (a) Big loads 20% fault time, (b) Big loads 40% fault time, (c) Big loads 60% fault time, (d) Big loads 80% fault time.

From Figure 7, we can also observe that among all the proposed FT mechanisms, Universal Logger mechanism has minimal space overhead when compared with other mechanisms. But considering the File Logger mechanism's minimal performance overhead, we can conclude that File Logger FT mechanism with bitbinary (Bit8 and Bit64) FT methods is the most suitable object-based FT mechanism.

D. RECOVERY TIME ANALYSIS

Minimizing the recovery time upon resuming from fault is one of the major objectives in our FT-LADS design. In this section, we have evaluated the FT-LADS recovery time for small and big workloads and compared it against the bbcp data transfer tool. For effective evaluation of recovery time of proposed FT methods, we created a simulation environment in which we generate faults after transferring 20%, 40%, 60%, 80% of total data size. As faults can occur at any end of the transfer, we can simulate the faults at either Source or Sink. However, for the purpose of our experiments, we have executed this simulation in the Source. Using the experimental environment described in Section VI, we measured the recovery time for both bbcp and FT-LADS. On these hosts, LADS uses CCI's Verbs transport, which natively uses the underlying InfiniBand interconnect and bbcp uses the IPoIB interface which supports traditional sockets.

In LADS, varying the number of I/O threads maximizes CPU utilization on the data transfer node. However, bbcp uses configurable window size and multiple streams to improve the performance. Based on the experimental results presented in LADS [1], [9], LADS data transfer performance increases linearly with the number of I/O threads. Whereas, bbcp has less impact while increasing the number of tcp streams. For fair performance comparison between the two, we have configured FT-LADS to use 4 I/O threads and bbcp to use 2 tcp streams with window size of 8MB. Our experiments are designed to calculate the transfer time before and after fault. Based on these times and using Equation 1, we estimated the recovery time.

Recovery times with all object based FT mechanisms and methods are compared with that of bbcp data transfer tool. Where we set LADS recovery time as the baseline for our experiments. Because the original LADS does not provide a resume operation, LADS has to transfer all the objects of the dataset upon resuming from faults. From the experimental results shown in Figure 8, 9 and 10, the later the fault occurs, the higher the recovery time is. Our aim is to minimize the impact of recovery time on the fault point. As per our logging mechanism, we delete the log file entries of the logical files, which are successfully transferred to the Sink. Due to this at any point of time, we are left with only those files which are currently being progressed. Thus, overhead for parsing FT log files to retrieve completed objects does not depend on the fault point.

The recovery time of File Logger mechanism at varying fault points for both big and small workloads is as shown in Figure 8 and 9. For the Transaction and Universal FT mechanisms, similar results were observed. In this paper, we only depict the results of File Logger FT mechanism.

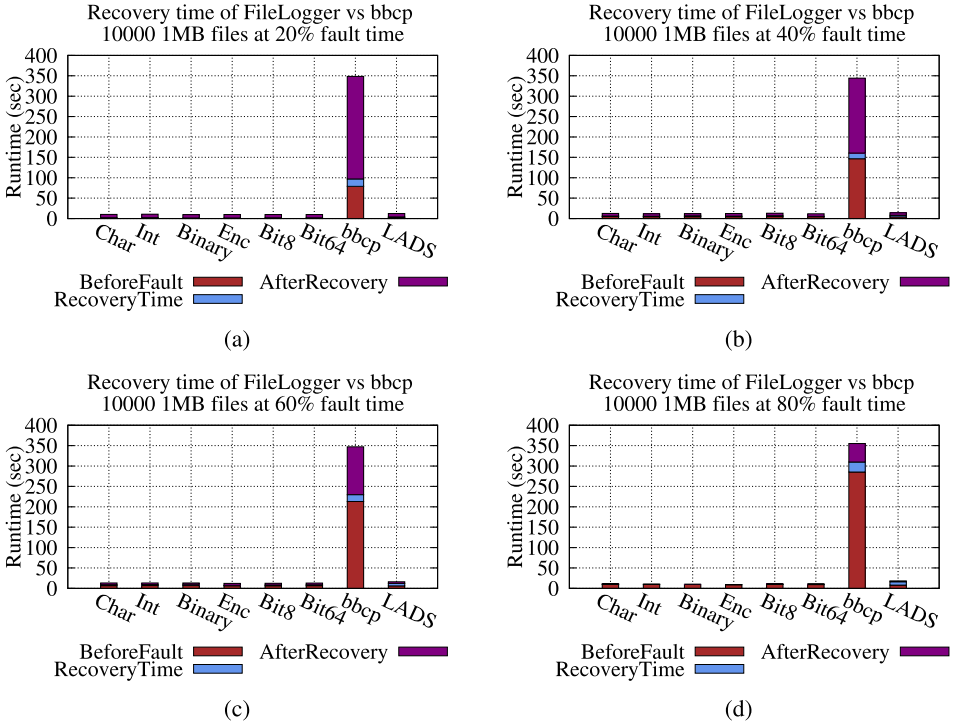


FIGURE 9. Recovery time analysis of FileLogger at varying fault timing for small workloads. (a) Small loads 20% fault time, (b) Small loads 40% fault time, (c) Small loads 60% fault time, (d) Small loads 80% fault time.

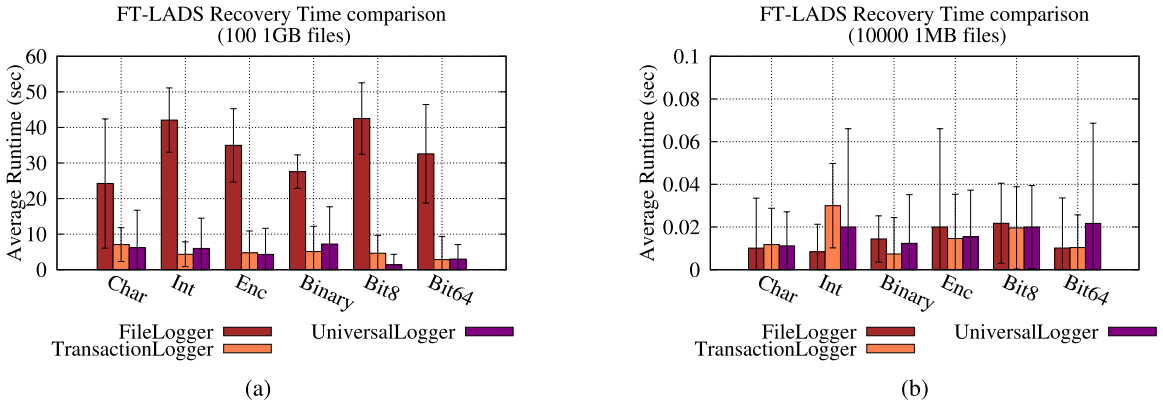


FIGURE 10. Recovery time analysis of FT Loggers at 80% fault timing. The 99% confidence intervals are shown in error bar. (a) Big loads 80% fault time, (b) Small loads 80% fault time.

1) BIG WORKLOADS

In the case of the File Logger mechanism, the recovery times for all FT methods exhibit similar recovery times irrespective of the fault points (Refer to Figure 8). Though the recovery time is much lower than LADS, all the methods of File Logger mechanism consume higher recovery times than bbcp. As bbcp FT is based on file offset, its recovery time is much less than that of File Logger. This is expected as with the File Logger mechanism, each logical file to be transferred is associated with one log file and while writing the logs to file, we just append the completed object index at the end of logger file. Due to this while retrieving the completed object information, an additional search overhead is involved.

For the Transaction and Universal Loggers, the recovery time overhead of big workloads is negligible. This is also expected because we sort the completed objects information per object index before writing to the log file. As mentioned in Section VI-B, we are using intermediate lists, which maintain the completed objects information of all files being transferred, by sorting based on the object index.

2) SMALL WORKLOADS

In contrast to the big loads, bbcp has a much higher transfer time for smaller workloads than LADS. Due to this, the recovery time overhead of FT-LADS is not directly comparable with bbcp. For quantitative comparison, the percentage of

recovery time relative to each method is calculated. At all given fault points, bbcp exhibits 5% to 7% recovery time overhead, while all the proposed FT methods experience around 12%-14% overhead.

Our small workload consists of files whose size is of 1MB which matches our transfer unit size. Due to this, a file transfer state can be either completed or transferred upon recovery from fault. There is no need to parse any log files upon fault and the proposed object based logger mechanisms just determine which files are already completed and start transferring the remaining files. As a result, we can conclude that with the proposed object based FT mechanisms, the recovery time overhead will not come into the picture.

In Figure 10, we have shown the recovery time comparison among the proposed FT mechanisms, considering 80% fault point as a reference, for both big and small workloads. As shown in Figure 10, we observe that for big workloads, the File Logger mechanism exhibits higher recovery time than the other proposed FT mechanisms. For small workloads, the recovery overhead for all mechanisms and methods are similar as shown in Figure 10 (b).

From Figure 10 (a) and 10 (b), we can observe that the Universal Logger mechanism exhibits lower recovery time upon fault. Also, among all the FT methods, bitbinary methods (Bit8 and Bit64) have minimal recovery overhead compared with the other FT methods.

Based on our evaluation results, the File Logger mechanism shows minimal impact on the performance while logging the completed object information. The Universal Logger is superior to other mechanisms with respect to recovery time upon fault. Also, among the proposed FT methods, the bitbinary methods (Bit8 and Bit64) have minimal space overhead and prove to have comparably lower recovery times among all the proposed FT mechanisms. Extending LADS with the Universal object based FT mechanism and bitbinary FT methods will improve data transfer performance in faulty environments.

VII. RELATED WORK

To meet the needs of big data transfers, prior studies have performed on the design and implementation of bulk data movement frameworks [16]–[18], [22]–[26]. GridFTP [18], which is an extended version of the standard File Transfer Protocol (FTP), provides high speed, reliable, and secure data transfer. The striping feature in GridFTP enables the support for multi-host to multi-host transfers. But this tool does not try to schedule the data transfer based on the underlying object locations. *bbcp* [16] is another data transfer tool which uses multiple streams for transferring large datasets. It uses a file based approach, which transfers the whole file data sequentially. XDD [22] optimizes the disk I/O performance by enabling file access with direct I/Os and using multiple threads for parallelism, and varying file offset ordering to improve I/O access times. RAMSYS [26], a resource-aware high-speed data transfer software, utilizes a multi-stage 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 and thereby offers improved scalability and resource utilization in modern multi-core systems. All these tools are useful for moving large data faster and secure from source host to remote host over the network, but none of them try to schedule based on the underlying object locations because they do not consider storage contention.

Another important aspect of these data movement frameworks is to resume data transfers after a fault. The GridFTP tool supports FT using restart markers (checkpoints). While transferring data, a GridFTP server automatically sends restart markers to the client. If the transfer has a fault, the client may restart the transfer by providing the markers received. The server will restart the transfer from the point where it left off based on the markers. GridFTP's Reliable File Transfer (RFT) service provides an interface to write the restart markers to a database so that it can survive a local fault. The *bbcp* tool employs FT mechanism based on checkpoint record. Upon initiating a new transfer, *bbcp* tool checks if checkpoint record of file being transferred exists or not. If record does not exist, it checks the target file attributes like name, size, etc. If they are identical with the Source file attributes, then *bbcp* assumes that the file is transferred successfully and skips the transfer. If file attributes are different, then it initiates a new transfer by creating a checkpoint record and transmit all the Source bytes to the target. Upon successful completion, it erases the checkpoint record. If checkpoint record exists, then it resumes the transfer by appending all untransmitted bytes to the target.

As all the aforementioned bulk data movement frameworks transfer the logical file data sequentially, it is possible to resume transfers using either a checkpoint based restart marker or offset record. Checkpoint based FT methods are light-weight and also possible to resume a transfer from restart marker or offset record without delay.

Our work focuses on entirely different scenario from the prior FT studies. In this work, we aim to support a resume functionality upon fault when the workload is transferred as objects rather than files, by exploiting the underlying storage architecture. Since a logical file is striped over multiple OSTs, it is possible to transfer one logical file's objects in random order. While the above mentioned checkpoint based restart marker or offset record is not sufficient to resume the transfer upon fault, our work proposes novel methods to handle FT in object-based big data transfers.

In our proposed object based FT mechanisms, objects which are successfully written to Sink PFS are marked as successful and we update the information of the object in the log file. Upon successful completion of all the objects of one logical file, the log information corresponding to the 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.

In object-based FT mechanisms, as all the objects of a file need to be logged, it involves access to the asynchronous filesystem API, which causes processing overhead. Also, it results in space overhead as all the object information is logged to the log file. This process also involves an additional overhead to retrieve the completed object information from the log file for resuming the transfer upon recovery from fault. Our solution proposes methods to overcome processing, space and recovery time overheads.

VIII. CONCLUSION

The LADS data transfer framework with its layout-aware and OST congestion-aware algorithms outperforms existing data transfer tools. But, the absence of FT support results in the data retransmission upon fault. As LADS employs object level scheduling algorithms, objects of one logical file may be transferred out of order, which makes traditional FT solutions based on logging file offset not suitable for LADS. In this work, we have implemented object-based FT mechanisms which can handle the out-of-order nature of object transmission. Based on the number of log files generated per dataset, we have proposed three different object logger mechanisms, *File Logger, Transaction Logger, and Universal Logger*. Also, we have proposed six different FT encoding methods: Char, Int, Enc, Binary, Bit8, and Bit64 to optimize the space overhead of these logging mechanisms. We have evaluated and compared the performance overhead of FT-LADS with LADS and concluded that proposed object based logging mechanisms do not negatively impact the LADS data transfer performance. Also, to evaluate the recovery time overhead of FT-LADS, we have created a simulation environment to generate faults at 20%, 40%, 60%, and 80% points of data transfer. From our evaluation results, we have observed that the recovery time in File Logger mechanism was two times higher than bbcp. However, the recovery time in the Transaction and Universal Logger mechanisms were considerably smaller than bbcp.

To conclude, the File Logger mechanism has minimal impact on logging the completed objects. The Universal Logger mechanism combined with bitbinary methods (Bit8 and Bit64) has a minimum overhead with respect to space and recovery time. With the addition of the proposed FT mechanisms, the LADS framework can provide both high performance and fault tolerance.

Acknowledgment

T. Kim was with the Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea.

REFERENCES

[1] Y. Kim, S. Atchley, G. R. Vallée, 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.

[2] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy server-side traces," in *Proc. 12th USENIX Conf. File Storage Technol. (FAST)*, Berkeley, CA, USA: USENIX Association, 2014, pp. 213–228.

[3] S. Oral *et al.*, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2014, pp. 217–228. doi: 10.1109/SC.2014.23.

[4] *Big Data*. [Online]. Available: https://en.wikipedia.org/wiki/Big_data

[5] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, pp. 60–65, Apr. 2011.

[6] A. Gulati, A. Merchant, and P. J. Varman, "pClock: An arrival curve based approach for QoS guarantees in shared storage systems," in *Proc. 7th ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2007, pp. 13–24. doi: 10.1145/1254882.1254885.

[7] B. Xie *et al.*, "Characterizing output bottlenecks in a supercomputer," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 8:1–8:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389007

[8] B. Welch *et al.*, "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

[9] Y. Kim, S. Atchley, G. R. Vallée, and G. M. 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.

[10] W. Chen and J. Tsa, "Fault-tolerance implementation in typical distributed stream processing systems," *J. Inf. Sci. Eng.*, to be published.

[11] A. Ebnenasir, "Software fault-tolerance," Dept. Comput. Sci. Eng., Michigan State Univ., East Lansing, MI, USA, Tech. Rep. [Online]. Available: http://www.cse.msu.edu/~cse870/Lectures/SS2005/ft1.pdf

[12] I. Koren and M. Krishna, *Fault-Tolerance Systems*. Amsterdam, The Netherlands: Elsevier, 2007.

[13] 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.

[14] 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. (ISPDC)*, Ljubljana, Slovenia, Oct. 2003, pp. 199–206.

[15] P. Kasu, Y. Kim, S. Park, S. Atchley, and G. R. Vallé, "Design and analysis of fault tolerance mechanisms for big data transfers," in *Proc. IEEE Int. Conf. Cluster Comput.*, Taipei, Taiwan, Sep. 2016, pp. 138–139.

[16] A. Hanushevsky. *BBCP*. [Online]. Available: http://www.slac.stanford.edu/~abh/bbcp/

[17] Y. Ren *et al.*, "Protocols for wide-area data-intensive applications: Design and performance issues," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 34:1–34:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389043

[18] W. Allcock *et al.*, "The globus striped GridFTP framework and server," in *Proc. ACM/IEEE Conf. Supercomput.*, 2005, pp. 54–64. doi: 10.1109/SC.2005.72.

[19] S. A. Atchley *et al.*, "The common communication interface (CCI)," in *Proc. HOTI*, Aug. 2011, pp. 51–60.

[20] *CCI: Common Communication Interface*. [Online]. Available: http://cci-forum.com/

[21] F. Wang, S. Oral, G. M. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding lustre filesystem internals," Oak Ridge Nat. Lab., Nat. Center Comput. Sci., Oak Ridge, TN, USA, Tech. Rep. ORNL/TM-2009/117, 2009.

[22] 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.

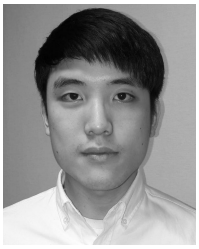
[23] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Design and performance evaluation of NUMA-aware RDMA-based end-to-end data transfer systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 48:1–48:10. doi: 10.1145/2503210.2503260.

[24] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda, "High performance data transfer in grid environment using GridFTP over InfiniBand," in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, May 2010, pp. 557–564. doi: 10.1109/CCGRID.2010.115.

[25] G. Vallée, S. Atchley, Y. Kim, and G. M. Shipman, "End-to-end data movement using MPI-IO over routed terabits infrastructures," in *Proc. 3rd Int. Workshop Netw.-Aware Data Manage.*, Nov. 2013, pp. 9:1–9:8.

[26] 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.

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



TAEUK KIM received the B.S. and M.S. degrees in computer science and engineering from Sogang University, Seoul, South Korea, in 2017 and 2019, respectively. He is currently a Researcher with TmaxCloud, Seongnam, South Korea. His research interests include parallel and distributed file systems and memory-level energy consumption.

JUNG-HO UM received the Ph.D. degree in computer engineering from Chonbuk National University, South Korea. He is currently a Senior Researcher with the Korea Institute of Science and Technology Information (KISTI). His current research interests include database, information retrieval, distributed computing, and multi-dimensional data analysis.

KYONGSEOK PARK is currently a Senior Research Scientist and a Research and Development Leader with the Scientific Data Technology Laboratory, Korea Institute of Science and Technology Information (KISTI). His current research interests include distributed computing, high-performance computing, numerical analysis, and machine learning.

SCOTT ATCHLEY received the B.S. degree in business administration and the M.S. degree in computer science from The University of Tennessee, in 1987 and 2002, respectively. In 2011, he joined the Oak Ridge National Laboratory as a HPC Systems Engineer. He was a Technical Staff Member with Myricom and a Research Leader with The University of Tennessee. He is currently the Team Leader of system architecture, resilience, and networking with the Technology Integration Group, ORNLs National Center for Computational Science. His research interests include high-performance interconnects and their interfaces, system architectures, and multi-level memories.

YOUNGJAE KIM received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2001, the M.S. degree from KAIST, South Korea, in 2003, and the Ph.D. degree in computer science and engineering from the Pennsylvania State University, USA, in 2009. From 2009 to 2015, he was a Research Staff Member with the Oak Ridge National Laboratory, USA. From 2015 to 2016, he was an Assistant Professor with the Department of Software and Computer Engineering, Ajou University, Suwon, South Korea. Since 2016, he has been an Associate Professor with the Department of Computer Science and Engineering, Sogang University. His research interests include operating systems, distributed systems, and file and storage systems, parallel I/O, emerging storage technologies, and performance evaluation.