2012 IEEE International Conference on Cluster Computing

Cx: Concurrent Execution for the Cross-Server Operations in a Distributed File System

Letian Yi, Jiwu Shu⊠, *JiaxinOu*, *YingZhao*
Department of Computer Science and Technology
Tsinghua University, Beijing, China
lonat.front@gmail.com, shujw@tsinghua.edu.cn

***Abstract*—Distributed metadata service is important for metadata intensive applications. Unfortunately, it leads to cross-server file operation, and maintaining the consistency of cross-server file operation creates a performance challenge because of *sequentially* executed sub-operations and costly *immediate commitment* among servers. In this paper, we observe that sub-operations can be executed concurrently and commitments can be delayed and batched for most cases in real applications, because the temporary inconsistency among servers rarely affects subsequent metadata operations. We propose a new protocol, Cx, in which the affected servers *Concurrently eXecute* the sub-operations of a cross-server file operation, and respond immediately to a client. Unless any sub-operation fails or other clients need to access the updated metadata objects, the commitment is delayed and batched with the other commitments. Evaluations of our Cx implementation in a parallel file system demonstrate Cx can significantly improve the performance of cross-server file operations, while retaining good scalability.**

I. Introduction

Metadata service has become more and more important for scaling the overall performance of large-scale data storage systems. Given tens of thousands of CPU cores per cluster, today's file systems on large-scale clusters are required to handle hundreds of thousands of files' metadata per second [1] [9] [10]. By judiciously distributing heavy metadata workloads among multiple servers, modern file systems split namespace and assign each part of the namespace to a different server to gain high concurrency. Accordingly, at many points in the namespace, a directory or a file must be assigned somewhere other than the home of its parent.

Because of the assignment, a file operation may be split into more than one *sub-operation*, each of which updates some metadata objects on a different server in cluster [11] [26]. For example, in supercomputing's checkpointing process, each process in cluster creates some files in a largely common directory that is normally managed by multiple servers to improve concurrency [9]; each creation requires two sub-operations: creating a new entry on one server that manages the parent directory, and creating a new inode on the other server that manage the newly created file. This file operation is termed as a *cross-server* operation. On the other hand, even with the distributed sub-operations of a cross-server operation, file systems must ensure the consistency, that is, file system should guarantee that applications either see the outcomes of all sub-operations of a cross-server operation, or none of them.

Although some protocols have been proposed to achieve the consistent goal for cross-server operations, they are at the sacrifice of performance, including using the high-overhead distributed transaction protocol [11][24], and migrating all related metadata objects into a single server to utilize the local consistent technique [26]. We note that, in all existing approaches (detailed in section 2), before the process can receive the response of a cross-server operation, the executions of assigned sub-operations are *serially* performed and the commitment must be achieved among affected servers; therefore, applications suffer from long cross-server operation delay because of the generally slow executions and expensive commitment.

In this paper, we propose Cx, a novel protocol to achieve the *optimal* performance of most cross-server operations in distributed file systems, while maintaining system consistency. In Cx, when a process (denoted as *P*) performs a cross-server operation, the affected servers *Concurrently eXecute* the sub-operations and leave the commitment being lazily performed after responding to the process. Due to lazy strategy, a large number of postponed commitments can be batched to explore disk bandwidth and reduce network interactions.

In the timespan between completing execution and achieving commitment of the cross-server operation, the related metadata objects updated by the operation may temporarily diverge, because one server may fail to execute its sub-operation. In this inconsistent timespan, when *the other processes* (the processes other than *P*) access the related metadata objects, the servers should immediately launch a commitment for the cross-server operation. This scenario is termed as a *conflict*. Although an immediate commitment is very costly in Cx, as shown in our analysis (detailed in section 3) on the characteristics of extensive and typical workloads, less than 4% cross-server operations of a process *conflict* with the operations of the other processes.

We have integrated the Cx protocol into OrangeFS [7], the next generation of PVFS2 [6]. Extensive trace-driven experiments demonstrate that, compared with the original

Table I: Typical split of the cross-server operations.

Ops	Sub-op on Coordinator	Sub-op on Participant
create	Insert a new entry in parent dir, and update parent inode	Adds an inode, set a flag to indicate it is a regular file
remove	Remove the file entry from parent dir, and update parent inode	Frees the inode if the nlink reaches 0
mkdir	Insert a new entry in parent dir, and update parent inode	Adds an inode, set a flag to indicate it is a directory, and allocate the entry space
rmdir	Remove the file entry from the parent dir, and update parent inode	Frees the inode if the nlink reaches 0
link	Insert a new entry in parent dir, and update parent inode	Increases the nlink of the file inode
unlink	Remove the entry from dir, and update parent inode	Decreases the nlink of the file inode

approach of OrangeFS, Cx can significantly improve the performance of cross-server file operations by more than 38%; meanwhile, benchmark-driven evaluations show that Cx can potentially improve the throughput of cross-server operations by up to 80%, while retaining the system's scalability on a cluster of 32 servers.

The rest of this paper is organized as follows. Background and related work are presented in Section 2. We describe the system model and design of Cx in Section 3 and 4, respectively. The recovery protocol of Cx is presented in section 5. We present the implementation details and performance evaluations of Cx in Section 6. The summary is described in Section 7.

II. Background

A. Problem Statement

We assume that multiple metadata servers as well as clients exist in a large-scale distributed file system. Previous studies have shown that metadata operations that need more than two servers are very scarce (less than 0.001%) [18][5][19], in this paper, we aim to optimize the *Cross-Server* operations that need *exactly two servers* [1].

In most distributed file systems, the affected servers of a cross-server operation are divided into a *coordinator* and a *participant*. Table 1 shows typical cross-server operations in distributed file systems and their sub-ops [11][24][26][7][13].

Our protocol of cross-server operations aims to achieve two goals:

- *Correctness*. Our protocol guarantees the correctness of a cross-server operation, i.e., a cross-server operation is guaranteed to be *atomic*. The whole system should either see the outcomes of all sub-ops of a cross-server operation, or none of them. Hence, the metadata cross servers are consistent after the execution of a cross-server operation.
- *Efficiency*. Our protocol tries to minimize the *process-centric* response time of a cross-server operation, i.e., the time between a process sends a cross-server operation request and receives a response of the operation.

[1]Operation that may require more than two metadata servers is *rename*.

B. Existing Approaches

We classify previous approaches along the execution order of all sub-ops, as shown in Figure 1.

Two Phase Commit protocol (2PC): In many file systems, such as Slice [13], IFS [14], Farsite [24], and DCFS [11], the correctness of a cross-server operation is ensured by a two-phase commit protocol (as shown in Figure 1(a)). Upon receiving a request from a client, the coordinator first initiates the first phase by sending a "VOTE" (i.e., a vote request) message to the participant, telling what sub-op the participant should perform. The participant executes its assigned sub-ops and sends the coordinator a message containing its own votes: "YES" or "NO", indicating the outcome of its executions. The coordinator collects the vote message and executes its sub-op, and then starts the second phase. If the coordinator successfully executes the sub-op and the vote is "YES", it sends a "COMMIT" messages to the participant. Otherwise, the coordinator sends a "ABORT" message to the participant that voted "YES". On receiving the message, the participant either commits or aborts its sub-op and sends an acknowledgement message to the coordinator. After receiving "ACK" messages, the coordinator sends a "RESP" message to the client indicating the end of the execution of the cross-server operation.

In the course of the execution, the servers record an operation log before sending a message out. Therefore, if any server fails, the coordinator can instruct participants to roll back their states to the beginning of the transaction.

Serially Execution protocol (SE): Some file systems, such as Diffs [15], PVFS2 [6] and OrangeFS [7], enforce an execution sequence of sub-ops. As shown in Figure 1(b), all sub-ops are serially and synchronously executed on the affected servers: the client first instructs the participant to execute its sub-ops; if the participant executes its sub-ops successfully, the client then asks the coordinator to execute its sub-op. If the coordinator fails to perform the assigned sub-op, the process withdraws the former sub-ops by sending a "CLEAR" message to the participant. However, if the client itself fails before sending the "CLEAR" message out, metadata across servers may be inconsistent, leaving orphan objects and violating the atomic property.

Centrally Execution protocol (CE): Ursa-Minor [26] shifts a cross-server operation to a single-server operation to centrally execute all sub-ops. As shown in Figure 1(c), when a cross-server operation is performed, all of the objects involved in the operation are migrated to the same server. The operation is then performed locally on that single server by reusing the server-side transaction techniques, such as journaling. The modified metadata objects are migrated back to the original server after completing the execution.

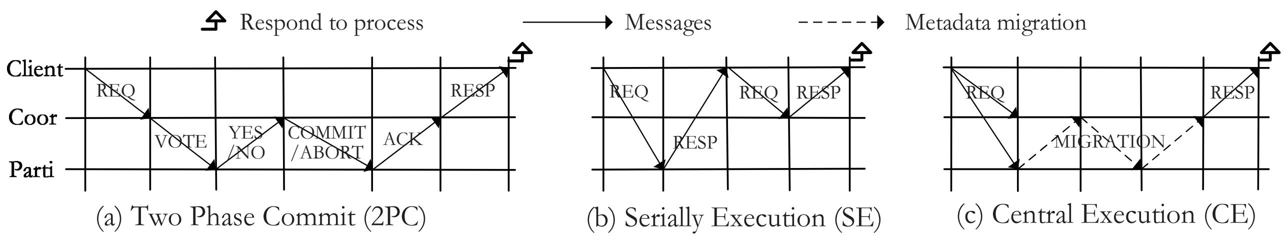


Figure 1: The communication patterns of existing protocols. The figure illustrates a example of operation affecting three servers. The *Coor* and the *Parti* indicates the *Coordinator* and the *Participant* in the protocols.

However, the previous studies [16] showed that the performance overhead for cross-server operations is high due to migration, about 7.5% slowdown of overall performance for the workload containing only 1% cross-server operations.

C. Characterizing Conflicts

We focus on the characteristic of *conflict ratio*, which is the proportion of the metadata operations that rise a conflict to all metadata operations in a workload. By our definition, all modified metadata objects by a cross-server operation may rise conflicts in the future.

Table 2 shows the total operations and conflict ratios in six typical workloads. The *CTH*, *s3d_fortIO*, and *alegra* traces are trace data for supercomputing applications collected on Redstorm cluster of Sandia National Laboratories [22]. They were collected while running CTH version 8.1, Fortran applications, and Alegra shock on 3300, 6400, and 5000 clients, respectively. The *home2*, *deasna-2*, and *lair62b* traces are taken from normal network server applications including primary home, research, and email directories of file servers in Harvard University, respectively, with more than tens of thousands of clients [20].

Table II: Conflict ratio in various workloads.

Trace	CTH	s3d	alegra	home2	deasna2	lair62b
Total Ops	505247	724818	404812	2720599	3888022	11057516
Conflict	0.112%	0.322%	0.623%	0.669%	2.972%	1.571%

From the table, we can find that the conflict ratio of all workloads is very low. For the traces collected from supercomputing clusters, the vast majority of files are generated by the periodical checkpointing jobs, in which every process of the application saves its own state into an individual state file. Since the states of a process are used by other processes only when a failure occurs, a state file is normally exclusively accessed by the process which created it and these files ingest most metadata operations in the traces. The exclusive accesses of state files will not rise any conflict, as conflicts can only occur on shared files.

The same trend is also observed from the traces of the network servers, in which the conflict ratios are slightly higher than those of supercomputing applications. In these applications, although the network servers provide the capacity of file sharing, users of network servers normally work in their own directories, exhibiting the exclusive-dominated access pattern. Recent studies on file access pattern confirmed similar observations as well [5].

Table III: Message representations.

Message	Signification	Src	Dest
VOTE	Queries the sub-ops' results	Coor	Parti
YES/NO	Indicates the execution results of a sub-op	Coor/ Parti	Pro/ Coor
COMMIT-REQ /ABORT-REQ	Asks to commit/abort the sub-ops' execution	Coor	Parti
ACK	Asks to complete a operation	Parti	Coor
L-COM	Asks to launch a commitment	Pro	Coor
ALL-NO	Denotes all executions of sub-ops have been aborted	Coor	Pro

III. Cx

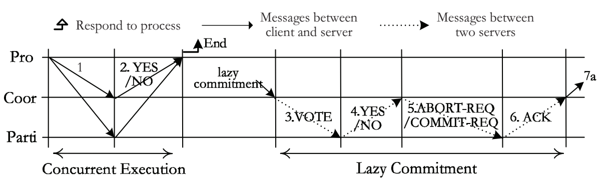
A. Notation

In Cx, each operation is uniquely identified by an *operation ID*, with three components: *a client ID, a process ID, an operation sequence number*, where the coalescence of a *client ID* and a *process ID* identifies a process in the cluster, and an *operation sequence number* assigned by the client node.

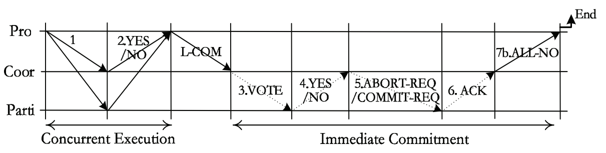
The notations of messages across servers and processes in Cx and their significations are listed in Table 3. The meanings of "VOTE", "YES/NO", "COMMIT-REQ/ABORT-REQ", and "ACK" messages are similar to those of 2PC protocol. Two new types of messages are introduced in Cx: a client process sends an "L-COM" message to ask the coordinate to launch an immediate commitment; and the coordinator sends an "ALL-NO" message to the client process to indicate all sub-ops have been aborted.

Similar to the existing approaches, Cx ensures consistency with the presence of node crashes by writing log records on affected servers. There are three types of records, all of which contains an operation ID to indicate the owner of a record:

- *Result_Record*: it contains the result of corresponding sub-operation at each server.
- *Commit_Record/Abort_Record*: it indicates all sub-ops' execution are successful/failed on the affected



(a) Gracious Execution.



(b) Execution with disagreement.

Figure 2: Protocol communication pattern for execution without conflict. The numbers refer to the main steps of the protocol numbered in the paper.

servers. For participant, it also indicates the whole cross-servers operation has been completed.

- *Complete_Record*: it appears only on the coordinator, indicating the whole operation has been completed.

B. Basic protocol without conflict

The discussion in this subsection assumes that only one process performs a cross-server operation (thus without conflict). In Cx, the execution of a cross-server operation consists an *execution phase* and a *commitment phase*. As described in Section 3.2, if all sub-ops reach an agreement at the client process, the commitment phase can be delayed. Otherwise, an immediate commitment must be performed to abort those sub-ops that have been executed successfully to regain a consistent system view.

We describe the main steps of Cx by tracing the messages in Figure 2, where we show all scenarios without conflict.

Execution phase

1. The process assigns sub-ops to each affected server.

2. On receiving the request, a server (either coordinator or participant) executes the assigned sub-op, and writes the execution result as a *Result-Record* into its local log. If the execution succeeds, the server sends a "YES" message to the process; otherwise, it responds with a "NO" message.

The process collects the responses from all affected servers. Based upon the consensus of all responses, there are two cases:

- All responses are in an agreement (either all "YES" or all "NO"): The process considers this cross-server operation completed now. However, from the perspective of the servers, this operation is still pending as the execution results of the sub-ops need to be committed. Therefore, lazily, a commitment is launched from the coordinator (as shown in Figure 2(a)).
- All responses are in a disagreement (containing both "YES" and "NO"): The process launches an immediate commitment by sending an "L-COM" to the coordinator (as shown in Figure 2(b)).

Commitment phase

3. When an "L-COM" request is received or a lazy commitment is launched, the coordinator starts a commitment by sending a "VOTE" message to the participant.

4. Upon receiving a "VOTE" message, the participant checks its *Result-Record*, and sends a "YES" or "NO" message to the coordinator according to its execution result of the corresponding sub-op.

5. On receiving the vote result, the coordinator decides whether the whole cross-server operation should be committed.

If the message is "YES" and the coordinator also has executed the assigned sub-op successfully, the coordinator writes a *Commit-Record* into its log, and then sends a "COMMIT-REQ" message to the participant.

Otherwise, if the participant has voted "NO", the coordinator writes an *Abort-Record* into its log and then sends an "ABORT-REQ" message to the participant.

6. Upon receiving an "ABORT-REQ" or a "COMMIT-REQ" message, the participant writes a *Commit-Record* or a *Abort-Record* for this operation respectively, and sends an "ACK" message to the coordinator.

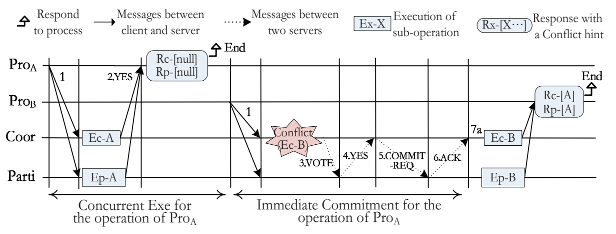
7. After receiving an "ACK" message, the coordinator acts differently in the following two cases:

- 7a. For a lazy commitment, the coordinator writes a *Complete-Record* for this operation (as shown in Figure 2(a)).
- 7b. For an immediate commitment, after writing a *Complete-Record*, the coordinator responds to the process with an "ALL-NO" message, which implies that all successful execution on affected servers have been aborted. When receiving the "ALL-NO" message, the process recognizes the whole cross-server operation as completed (as shown in Figure 2(b)).

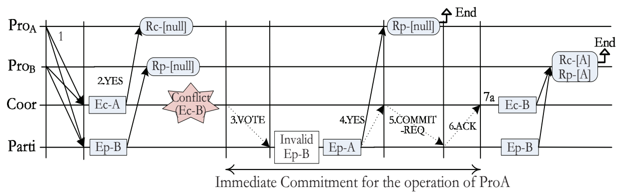
The design of concurrent execution without conflict is based on the principle that the metadata operations of a process are performed synchronously. However, this consistent state is not hold for the processes other than P in the cluster. This is because the lazy commitment may leave some active objects that are not achieved agreement among the affected servers, e.g., a server may have written the sub-op's *Result-Record* of a cross-server operation and the other has not. As a result, if the processes other than P access the active objects, they impose conflicts. We next discuss how to extend the basic protocol to handle these conflicts.

C. Handling Conflicts Using Hint

Formally, we classify conflicts into two categories: (1) *ordered* conflicts, in which every server sees the same arrival sequence of the sub-ops from different processes, and (2) *disordered* conflicts, in which a server sees a different arrival


Respond to process
Messages between client and server
Messages between two servers
Ex-X Execution of sub-operation
Rx-[X⋯] Response with a Conflict hint
ProA
ProB
Coor
Parti
1
2.YES
Rc-[null]
Rp-[null]
End
Ec-A
Ep-A
Conflict (Ec-B)
3.VOTE
4.YES
5.COMMIT -REQ
6.ACK
7a
Ec-B
Ep-B
Rc-[A]
Rp-[A]
Concurrent Exe for the operation of ProA
Immediate Commitment for the operation of ProA

(a) Ordered Scenario.



(b) Disordered Scenario.

Figure 3: Protocol communication pattern for execution without conflict. The numbers refer to the main steps of the protocol numbered in the paper.

sequence of the sub-ops from that of the other server. Cx resolves the disordered conflict by enforcing the execution order of the coordinator to the participant. That is, upon detecting a conflict, the coordinator instructs the participant to obey the same execution order of the sub-ops with the coordinator if the participant sees a different arrival sequence of the sub-ops. Toward this end, the participant is required to invalidate the execution of a executed sub-op; besides, since the process may have already received the response of a invalidated execution, it must be able to distinguish the response of the invalidated execution.

How Cx handles the ordered conflict is straightforward. As illustrated in Figure 3(a), when the coordinator receives the sub-op request of Pro_B (*Ec-B*), it detects a conflict because the sub-op accesses the active objects of a pending operation of Pro_A, and an immediate commitment for the pending operation is launched; after the pending operation is agreed on the affected servers (at step 7a), the operation of Pro_B is then safely executed.

Different form the basic protocol, to help a process distinguishing a response of invalidated execution in the disordered conflict scenario, the server constructs a *conflict hint* for each response. Given a sub-op `SOP`, if it raises a conflict with a sub-op `SOP'` and `SOP'` must be committed before executing `SOP`, the conflict hint for `SOP`'s response is constructed as *[SOP']*; otherwise, the hint is built as *[null]*. For example, in the step 2 of Figure 3(b), the coordinator responds to the Pro_A with a conflict hint of *[null]*, because the sub-op of Pro_A (*Ec-A*) does not raise a conflict with the other sub-ops. With the help of conflict hint, a process recognizes a cross-server operation as complete *only when* it has received the responses from both affected servers with the same conflict hint.

We next track the steps in Figure 3(b) to explain the details of how Cx leverages conflict hint to handle the disordered conflict.

1. The process assigns the sub-ops to the affected servers.

2. On receiving a sub-op request, the server checks the conflict.

If the sub-op does not access the active objects, the server executes the sub-op and writes a *Result-Record* into log. Then the server responds to the process with a “YES” or a “NO” according to the result of a execution. In Figure 3(b), since both *Ec-A* and *Ep-B* are executed on the servers without raising any conflict, Pro_A and Pro_B receives a response with a *[null]* conflict hint.

Otherwise, if detecting a conflict, the server blocks the sub-ops, and the coordinator launches an immediate commitment for the pending operation. In Figure 3(b), both *Ec-B* and *Ep-A* are blocked on the servers, because they tend to access the active objects.

3. The coordinator launches an immediate commitment for the pending operation by sending a “VOTE” message to the participant. The message also implies that the coordinator tends to instruct the participant to obey its execution order.

In Figure 3(b), the coordinator wants the participant to execute *Ep-A* before *Ep-B*.

4. Upon receiving the “VOTE”, the participant obeys the coordinator’s execution order.

In Figure 3(b), the participant first invalidates the execution of *Ep-B* by invalidating the *Result-Record* of *Ep-B*, and then executes *Ep-A*. After executing *Ep-A*, the participant writes a *Result-Record* for *Ep-A* and responds to the Pro_A according to the execution result. Since *Ep-A* does not raise any conflict, the response contains a *[null]* conflict hint. The invalidated *Ep-B* is re-queued as a new arrival sub-op request.

At this point, because Pro_A has already received the responses with the same conflict hint (*[null]*) from each server, its operation can be recognized as complete.

5. Like the basic protocol, upon receiving a vote result of the participant, the coordinator writes a *Commit-Record*/*Abort-Record* and sends a “COMMIT-REQ”/“ABORT-REQ” according to the vote result.

6. Like the basic protocol, on receiving an “ABORT-REQ” or a“COMMIT-REQ” message, the participant writes a *Commit-Record* or a *Abort-Record*, and sends an “ACK” message to the coordinator.

7a. On receiving the “ACK”, the coordinator writes a *Complete-Record* for the operation of Pro_A.

In Figure 3(b), after the servers have achieved agreement on the operation of Pro_A, they can safely execute *Ec-B* and *Ep-B*, and writes a *Result-Record* for the execution. Each response for *Ec-B* and *Ep-B* contains a conflict hint of *[A]*, indicating that there was a conflict on the operation of Pro_A.

D. Operation Logs and Recovery

Similar to traditional approaches, all log records in Cx must be synchronously written into the persistent storage, such as a log file. However, when the log becomes full, a server must block the new-arrival sub-op requests and perform *pruning*. Toward this end, in Cx, the log records are periodically pruned after the commitments are performed to avoid blocking request. For the coordinator, if a Complete-Records is presented in the log, all log records of that operation can be pruned; for the participant, on the other hand, a presented Commit-Record/Abort-Record indicates that all log records of that operation can be pruned.

The recovery process for node starts when the failure detection subsystem confirms a crash on any node. After a crashed server reboots, it informs all other collaborating servers to go into the recovery state, in which the rebooted node reads its log and recovers according to the on-disk log records. In the recovery process, the whole file system stops responding new requests.

The main idea of our recovery protocol is to resume all half-completed commitments of cross-server operations left in the log file on a server before it crashed. Our recovery protocol specifies the behaviors for the operations that left at least a record on the crashed server. From the *Result-Record* of an operation, the rebooted server can determine whether it is the coordinator of that operation. Depending on its role, the resumption of an operation varies on different rebooted servers.

IV. Evaluations

A. Implementation

We have implemented Cx in the OrangeFS (OFS) [7] to verify its feasibility and effectiveness. The OFS [7] comprises three major components: (1) *metadata servers* (MDSs) are mainly responsible for file metadata. The metadata is stored as rows in Berkeley DataBase (BDB). (2) *IO servers* are actual storage depositories for file data which are stored in files in a local directory tree. (3) Applications are run on *clients*. In OFS, to create a new file, a directory entry is assigned to a server *based on its name hash value*, and the file's metadata object (inode) is *randomly* created on one server in the cluster.

Log organization: Cx needs to write log records for fault management activities. Log records can be stored in the BDB or can be organized as a log-structured file. We choose the latter approach to exploit more disk bandwidth, and build an index on top of it to accelerate searches.

Batched commitments: The permitted lazy commitments are batched and launched by triggers. Our implementation currently supports two types of triggers: (1) *Timeout* trigger, (2) *Threshold* trigger. The timeout trigger fires if a certain period of time has elapsed since the last commitment, and the threshold trigger fires when the number of pending operations goes beyond a threshold since the last commitment. Some alternative triggers can be used to react to the workload characteristics, such as system idle time, we pursue them in our future works.

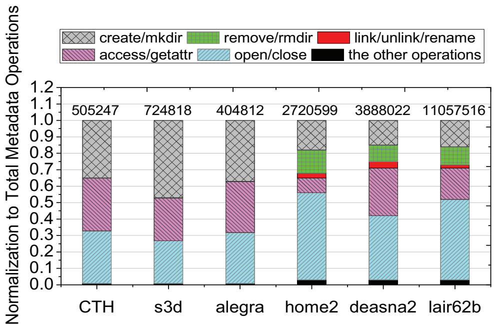


Figure 4: Metadata operations distribution in workloads.

B. Experiments setup

All experiments were performed on a cluster up to 32 metadata servers, and the number of load-generating clients is four times of that of servers. Each machine is equipped with dual quad-core 2.83GHz Intel Xeon processors, 8GB memory and a 10 GigE NIC, connecting by Catalyst-3750 10 GigE switches. All nodes were running the Linux 2.6.27.5-117 kernel (fedora 10 release). Every metadata server of OFS stores metadata (database is built on a local ext3 file system) on one 7200rpm SATA disk.

The traces used in our experiments are described in section 2. The metadata access patterns in term of the distribution of various operations are shown in Figure 4 for all traces, with the total number of operations written on top of each bar.

We used the *Metarates* benchmark in our benchmark-driven experiments, which ia an MPI application and used widely by distributed file system vendors [8]. We emulated two typical workloads using Metarates: (1) a *read-dominated* workload, which consists of 20% updates and 80% stats, since Vogels found that 79% accesses to files were read-only in many file systems [18]; (2) a *update-dominated* workload, which consists of 80% updates and 20% stats [10]. To evaluate the large directory service with Cx, the update and stat operations in these workloads are designed to *concurrently create/remove* zero-bytes files in a common directory, and to *concurrently stat* the generated files, respectively.

Since we used the Metarates benchmark to measure the aggregated throughput for various file systems, we make sure that enough files are created on each server to reach its peak performance. As OFS randomly distributes sub-files in a directory on servers, the number of files on each server is about the same on average. In our experiments, a single server manages 40,000 files in a directory, and 32 servers manage up to a 1.28 million files in a directory.

Unless specified otherwise, we conducted our experiments with the log size of 1MB for each server, and employed

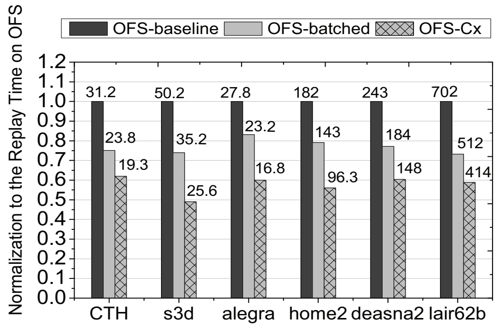


Figure 5: Trace-driven evaluation.

the timeout trigger for periodical lazy commitments with a timeout value of 10 seconds.

C. Overall evaluations

To more accurately quantify the performance improvement of cross-server operations, we compare our Cx implementation with two baseline systems: (1) OFS and (2) OFS-batched. Similar to OFS, in OFS-batched, the sub-ops of a cross-server operation are serially performed on affected servers; however, instead of synchronously writing the updated objects into BDB for every sub-op, the updated objects are logged and the batched modifications are lazily flushed into BDB. OFS-batched is used to answer the question: how much improvement can Cx gain from the batched write-back?

1) Trace-driven evaluations: Our first set of experiments are to compare OFS, OFS-batched, and OFS-Cx by relaying six traces, using 4 and 8 metadata servers. Figure 6 illustrates the relative performance in terms of trace replay time (in seconds), using 8 servers. The similar trend can be observed when using 4 servers and we omit its figure for page limit considerations.

As shown in Figure 5, OFS-Cx significantly speeds up the replay time across six traces, with an improvement of at least 38%. The observed improvement is related to the proportion of cross-server operations in a trace. For example, due to the fact that about 48% of metadata requests are cross-server operations, greater improvement is achieved on the *s3d* trace, with an improvement of more than 50%; on the *CTH* workload, on the other hand, OFS-Cx improves the performance about 38%, with about 35% cross-server operations.

The reasons why OFS-Cx achieves improvements on cross-server operations are twofold. First, the sub-ops of a cross-server operation are concurrently executed on affected servers. Therefore, the response time is significantly improved by reducing the time cost of serial executions. This substantial improvement can be observed from the fact that OFS-Cx outperforms OFS-batched by at least 16%. Second, since the synchronization for modifications on metadata objects is no longer performed for every operation in OFS-Cx, submitting batched modifications into BDB increases the possibility of merging disk requests in kernel's IO scheduler, decreasing the number of disk accesses [25]. The effectiveness of batched submission is demonstrated by the gains of a batched-version of OFS, with an improvement over OFS by at least 15% in our experiments.

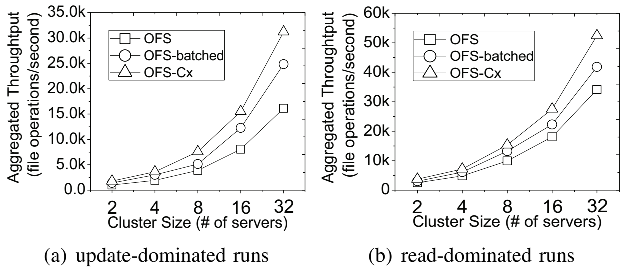


Figure 6: Benchmark-driven evaluations.

Table IV: Messages (in million) generated in the trace replays and the message overhead of OFS-Cx.

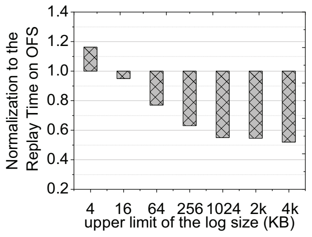
	CTH	s3d	alegra	home2	deasna2	lair62b
OFS	1.261	1.963	0.624	5.641	7.815	22.616
OFS+Cx	1.288	2.021	0.630	5.815	8.002	23.136
Overhead	2.2%	3.0%	1.0%	3.1%	2.4%	2.3%

We also measured the message overhead of Cx in the runs. Table 4 shows the number of messages in trace replays in million of messages of OFS and OFS-Cx. Theoretically, OFS-Cx would send more messages than OFS for completing a commitment among affected servers as analyzed in section 4.4. However, the actual additional cost is very low at less than 4% as shown in Table 4, because lazy commitments can send batched messages, reducing the number of messages greatly. The message overhead increases as the conflict ratio of a workload increase, because more immediate commitments have to be performed and become a big factor, which is examined carefully in section 6.4.

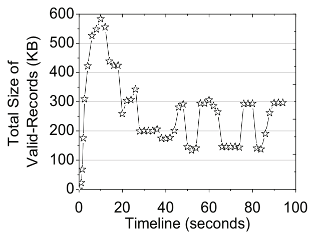
2) Benchmark-driven evaluations: In addition to trace-driven experiments, we also conducted experiments to examine the throughput of various version of OFS when scaling the cluster size, driven by the Metarates benchmark. In these experiments, our configuration uses 8 processes per client.

Figure 6 plots the aggregated throughputs of OFS, OFS-batched, and OFS-Cx, in file operations per second. Similar to the result of trace-driven experiments, we can see that OFS-Cx outperforms OFS-batched and OFS in terms of operation rate in benchmark experiments, and, the aggregated throughput of OFS-Cx scales well when increasing the number of servers up to 32. OFS-Cx gains an improvement of at least 40% and 70% on read-dominated and update-dominated experiments, respectively, which is not surprising, as the update-dominated workload contains more cross-server operations.

Note that Cx gains an improvement of 82% on the update-dominated workload with 8 servers, whereas the



(a) impact of upper-limit.



(b) valid-records' size

Figure 7: Sensitivity of log size.

improvement on trace-driven experiments with 8 servers is between 38% to 50%. In addition to the reason of larger proportion of cross-server operations in the update-dominated workload, other key reason lies in the fact that, the update-dominated workload accesses files in a single directory, whose metadata objects are sequentially placed on disk in OFS [7]. Consequently, batched updates on these objects may constantly push the performance of BDB write-back close to its peak point.

D. Sensitivity study

Cx's performance is likely to be influenced by several parameters, including the log size, conflict ratio, and batched commitment strategies. We study the sensitivities of these parameters in this subsection. For page limit considerations, we limit our study of these parameters to the *home2* trace, and other traces show similar trends. Unless specified otherwise, we conducted our experiments on 8 servers in this subsection.

1) Log size: As described in section 5.1, when the log file is already full, the new-arrival requests are blocked until some records have been pruned from the log file, therefore it is meaningfully to evaluate how the upper-limit of log size affects the performance of the cross-server operations. We conducted experiments that measure the throughput as a function of the upper-limit of log size. We define a *valid-record* on a server as the record that relates to an operation that needs to be lazily committed. We also measured the size variance of all *valid-records* size during the runtime without restricting the upper-limit of log size, to see how large a log file needs to be.

Figure 7(a) shows that OFS-Cx improves the performance more significantly with a larger maximum log size on each server than that with a smaller one. The reason is that when the upper-limit is smaller, the log is more likely to be filled during the runs; to prune the log records, the coordinators needs to launch commitments for operations and synchronize metadata objects into database; as a result, more blocked requests suffer from long delays. Figure 7(b) further illustrates the size variance of valid-records during the replay. We can find that the size of valid-records increases rapidly in the initial seconds (about the first 10 seconds), with the peak value of 600KB, and then decreases. The increase is

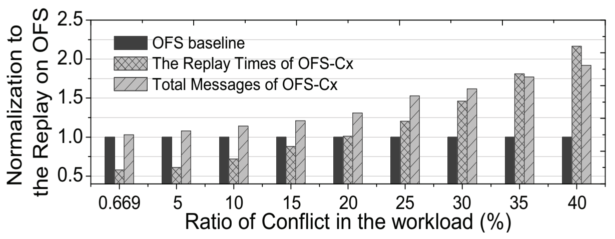


Figure 8: Impact of conflict ratios.

because that in the initial seconds cross-server operations are graciously executed without commitments; after that, lazy commitments are launched and valid-records are pruned. Since we employed a timeout trigger of 10 seconds, periodical drops can be seen from figure 7(b).

2) Conflict Ratio: Since the design of Cx is motivated from our analysis on low conflict ratios in various workloads, it is interesting to examine the impact of conflict ratios on Cx. In order to emulate different conflict ratios, we injected some *lookup* requests to add some immediate commitments for cross-server operations in the *home2* trace.

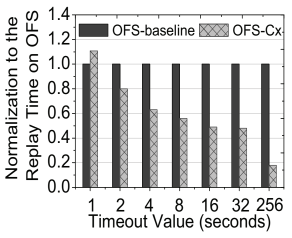
Figure 8 shows the replay time and the message cost of OFS-Cx when increasing the conflict ratio in the *home2* trace, where 0.669% is the original conflict ratio. As expected, because immediate commitments for cross-server operations increase not only network messages but also the number of individual log writes on each server, the throughput decreases as the ratio increases. Nevertheless, as long as the conflict ratio is lower than 20% in our experiments, OFS-Cx outperforms OFS.

3) Batched commitment strategies: We also examined the sensitivity of Cx to the batched commitment strategies for gracious executions. We adopted two strategies for lazy commitments: timeout and threshold. To accurately investigate the impact of these strategies themselves, we unlimited the upper-limit of log size.

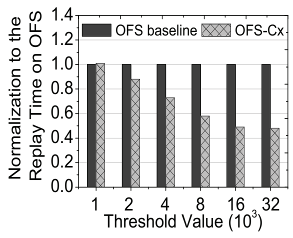
Figure 9 shows the replay times of *home2* using two strategies for lazy commitments. From the figure, one can see that the replay time decreases as the value of timeout or threshold increases. The reason is that a higher timeout or threshold value for lazy commitments implies more delayed commitments to be handled together, so that the kernel's scheduler has more opportunities to merge disk requests with more batched requests. However, if setting a high value, consequently the number of valid records on the log file increases as well, thus prolonging the recovery time potentially. Note that, when the timeout value is too large so that no lazy commitments are actually triggered during the trace replay, OFS-Cx reaches its optimal performance (as shown in Figure 9(a) with a timeout value of 256 seconds).

E. Recovery

To evaluate how the recovery of Cx affects the system availability, we conducted experiments on 8 servers to examine the time of recoveries. To simulate a server crash,



(a) timeout sensitivity



(b) threshold sensitivity

Figure 9: Sensitivities of Batched Approaches.

we killed the processes on a server after it has accepted a specific size of valid-records.

Table V: Recovery time as increasing the valid-records' size.

Valid-Records' size (KB)	5	10	50	100	500	1000
Recovery Time (seconds)	3	6	8	10	12	17

Table 5 shows the recovery time for different sizes of valid-records. From the figure, one can see that the recovery time of OFS-Cx becomes larger when the size of valid-records increases, because more cross-server operations need to be committed and metdata objects to be submitted to BDB during the recovery process. However, when the size of valid-records increases 100 times (from 10 KB to 1000KB), the recovery time of OFS-Cx increases less than 3 times, and the recovery time is within tens of seconds.

V. Conclusion

In this paper, we observe that sub-operations of most cross-servers operations can be executed concurrently and commitments can be delayed and batched for most cases in real applications because the temporary inconsistency among servers rarely affects subsequent metadata operations.

Based on this observation, we propose a new protocol, Cx, to separate the expensive commitment from execution for most cross-servers operations, taking the advantages of batched commitments. Our evaluations of a Cx's implementation demonstrate that this separation significantly improves the performance of cross-servers operations than the traditional approaches.

Acknowledgment

This work was supported by the National Science Foundation for Distinguished Young Scholars of China under Grant No.60925006, the National Science and Technology Support Plan of China under Grant No.2011BAH04B02, and the National High Technology Research and Development Program of China under Grant No.2012AA011003£

References

[1] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C., *Ceph: A scalable, high-performance distributed file system*. In OSDI'06.
[2] http://www.lustre.org/documentation.html.
[3] Welch, B., Unangst, M. Abbasi, Z. Gibson, G. AND Mueller, B. *Scalable performance of the panasas parallel file system*. In FAST'08, 2008, 17-33.
[4] F. Schmuck and R. Haskin. *GPFS: A shared-disk file system for large computing clusters*. In FAST'02, pages 231-244.
[5] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, Ethan L. Miller, *Measurement and analysis of large-scale network file system workloads*, USENIX ATC, 2008, 213-226.
[6] http://www.pvfs.org/
[7] http://www.orangefs.org/
[8] www.cisl.ucar.edu/css/software/metarates/
[9] Swapnil Patil and Garth Gibson. *Scale and Concurrency of GIGA+: File System Directories with Millions of Files*. In FAST'11, San Jose, CA, 2011.
[10] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. *PLFS: A Checkpoint Filesystem for Parallel Applications*. In SC'09, November 2009.
[11] J Xiong, Yiming Hu, Guojie Li, Rongfeng Tang, and Zhihua Fan. *Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems*, In IEEE TPDS, Volume 22 Issue 5, May 2011.
[12] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[13] D.C. Anderson, J.S. Chase, and A.M. Vahdat, *Interposed Request Routing for Scalable Network Storage*. ACM Trans. Computer Systems, vol. 20, no. 1, pp. 25-48, Feb. 2002.
[14] M. Ji, E.W. Felten, R. Wang, and J.P. Singh, *Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services*, Proc. Fourth USENIX Windows Systems Symp. Aug. 2000.
[15] Z. Zhang and C. Karamanolis, *Designing a Robust Namespace for Distributed File Services*, Proc. 20th IEEE Symp. Reliable Distributed Systems, pp. 162-171, Oct. 2001.
[16] S. Sinnamohideen, R. Sambasivan, J. Hendricks, L. Liu, and G. Ganger, *A Transparently-Scalable Metadata Service for the Ursa Minor Storage System*, In USENIX ATC' 10, June 2010.
[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*. In SOSP' 03, pp.29-43, 2003.
[18] D. Roselli, J. Lorch, and T. Anderson, *A Comparison of File System Workloads*. In USENIX ATC'00, pp. 41-54,
[19] Werner Vogels. *File System Usage in Windows NT*. In SOSP'09, pp.93-109, 1999.
[20] Daniel Ellard, Jonathan Ledlie, Pia Malkani, Margo Seltzer. *Passive NFS Tracing of Email and Research Workloads*. In FAST'03, pp. 142-155.
[21] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. *A Trace-Driven Analysis of the UNIX 4.2 BSD File System*. In SOSP'85.
[22] http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/
[23] Los Alamos National Labs. Roadrunner. http://www.lanl.gov/roadrunner/, Nov 2008.
[24] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, etc. *FARSITE: Federated, available, and reliable storage for an incompletely trusted environment*, in OSDI'02.
[25] Sitaram Iyer and Peter Druschel. *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*. in SOSP'01, 2001.
[26] M. Abd-El-Malek, et al. *UrsaMinor: versatile cluster-based storage*. Conference on File and Storage Technologies. USENIX Association, 2005.