Crash-Proofing uFS: Achieving Resilience in User-Space File Systems with Client Retries

Benita Kathleen Britto
bbritto@wisc.edu

Abstract— **The uFS paper proposes a semi-microkernel approach to file system design, which enables high-performance while allowing user-level development and deployment. However, if the file server crashes, the system lacks the capability to retry the operations that were invoked during the crash. One solution is to retry the operations on the client-side, but this approach has challenges such as losing unpersisted data and open file descriptor states. In this report, we explore a solution to this challenge by implementing an idempotent version of uFS and allowing the server to asynchronously notify the client of persisted writes. This enables the client to resend unpersisted-write operations on server restart, achieving higher performance with no lost data. While there are significant challenges to overcome, this approach requires minimal changes to the uFS server.**

I. Introduction

The uFS paper [6] proposes a novel approach to file system design that is based on a semi-microkernel architecture. This approach is similar to the one used in high-performance networking systems and it allows for the hoisting of most of the file system functionality out of the kernel, leaving other OS functionality in the main monolithic OS.

The uFS system is designed to achieve high performance under varying application demands while enabling the benefits of user-level development and deployment. The uFS library is carefully designed for performance, with various optimizations like metadata journaling, where writes are performed in memory unless a dirty ratio is hit or flush is called explicitly by the client. Additionally, as per metadata journaling, updates are made on on-disk structures through the checkpointer invoked before starting the uFS server. This makes uFS a fully crash-consistent user space file system.

While uFS offers many benefits, there is one major drawback. If the file server crashes, there is currently no capability to retry the operations that were invoked during a crash. To tackle this problem, one approach is to retry the operations on the client-side. However, this approach has its own set of challenges.

One of the major challenges is that the server loses not only all unpersisted data but also the state for open file descriptors. This means that clients must be modified to track their outstanding operations. When clients reconnect to a restarted server, they need to retry operations that did not complete. Additionally, we need to ensure that the server is stateless and to use an idempotent API between the client and server, as in NFSv2 [7]. However, handling write operations in this approach can result in lost data. If a client does not wait for writes to be persisted through the server, it may lose data when the server crashes. On the other hand, if the client waits for the write to be persisted, it will obtain poor performance.

To overcome this challenge, the idempotent protocol can be extended so that the server asynchronously notifies a client of persisted writes. This way, the client can resend the unpersisted-write operations on server restart and obtain higher performance with no lost data. However, this requires significant modifications to the client and server.

In summary, while the uFS system offers many benefits, there are also significant challenges when it comes to handling crashes. The approach of retrying operations on the client-side has significant drawbacks, while using an idempotent API can result in lost data. However, by extending the idempotent protocol and making significant modifications to the client and server, it is possible

to achieve client retries with minimal changes to the uFS server. We detail how client-retries are achievable over uFS in this report.

II. Background

The following is a summary of important aspects of uFS, namely its architecture, data copy elimination, crash consistency, and load balancing. Understanding these aspects is crucial for implementing client retries.

A. Architecture

The uFS system consists of a user-level process (uFS Server) and a library (uLib) linked with each application. The uServer interacts with the storage device using NVMe commands and pinned memory for data transfer. It is POSIX-compliant but does not support extended attributes, links, mmap, and chmod/chown. The system uses a per-application thread-safe lockless ring buffer for control transfers and thread-private memory shared with uServer for data transfers. The server performs five tasks, which include receiving requests, processing requests, attending to background activity, initiating device requests, and notifying clients of results. uFS is designed for multi-core systems and utilizes data parallelism by dividing filesystem data structures across cores in a shared-nothing architecture. Each worker server thread has its own ring buffer, ready queue, and qpairs with the storage device, and the server process can be composed of multiple threads.

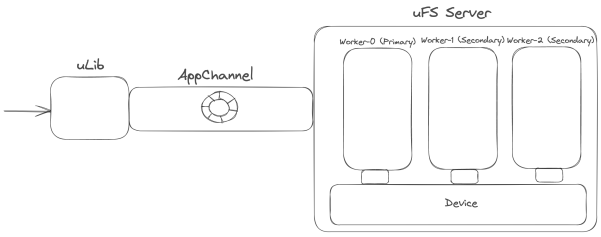

Fig. 1. Simplest view of the uFS system

B. Data copy elimination

To eliminate data copying between applications and uLib, uFS provides a shared memory region that can be accessed directly by applications. uFS malloc allocates from this shared space, which can either be exposed to applications for maximum performance or kept within uLib for portability. By using a buffer from uFS malloc and passing it to uFS allocated write, an application can avoid copying data between itself, uLib, and uServer. Alternatively, if an application calls uFS write(buf), uLib calls uFS malloc and copies the contents of buf to shared memory before calling uFS allocated write. This reduces latency by avoiding the extra copy.

C. Crash consistency and load balancing

To achieve crash consistency, uFS uses ordered metadata journaling and logical journaling. The primary goal of uFS is to allow each thread to write to a shared journal with minimum coordination to achieve high performance. It also includes a load management feature that dynamically adapts the number of cores dedicated to the server and balances the allocation of inodes across those cores as a function of the current workload. The load manager thread wakes periodically to gather load statistics from each worker, decide on the number of cores to use in the next window, and direct the workers to perform load balancing. The manager tries to minimize both the number of cores and the queuing time of each request by keeping each below a configurable threshold. At the end of each balancing window, the manager determines the amount of per-client load to shift from each overloaded worker to each under-loaded worker.

D. Server worker threads

The primary server worker thread owns all directory inodes and assigns file inodes dynamically across worker threads based on load. Despite its high performance and load balancing capabilities, uFS does have some limitations, such as not supporting extended attributes, links, mmap, and chmod/chown and does not support retries. Nonetheless, it remains a highly efficient and effective file system for multi-core systems.

III. Design

In our efforts to incorporate client retries into uFS, we went through multiple design iterations. Our final design included making uFS idempotent

and tracking pending requests. Initially, we considered five versions of the design, each with a single threaded client and server. However, we later expanded our approach to support multithreaded clients and servers.

A. Design v1: Adding idempotency

In Design v1, we start by making the API idempotent. To achieve this, all non-path based APIs that rely on file descriptors need to be changed because on a server crash, the server loses the context of open file descriptors, making it impossible for the client to use them when the server comes back up. Therefore, the uFS server returns the inode number instead of a file descriptor to eliminate any file descriptor state on the server, as shown in Fig 2.

To enable client retries, the client places a request in the ring buffer, and the server picks it up and operates on it. When the client places the request in the ring buffer, it sets the status of the message as `READY_FOR_SERVER`. The server changes the status to `IN_PROGRESS` when it services the request and then changes it to `READY_FOR_CLIENT` when it has finished processing the request. The client polls for the status of the request to change to `READY_FOR_CLIENT` when it places it in the ring buffer, and this call is blocking. In the event of a server crash, the client will poll indefinitely on the message until it detects that the server is unavailable.

To detect server unavailability, the client places the message in the ring buffer and polls the message five times before it determines that the server is unavailable. The client then declares the server as unavailable. However, we acknowledge that this is a naive approach to detecting server unavailability and suggest that a better approach could be implemented in the later sections. Once the client detects that the server is unavailable, it can invoke its retry process, where it retries all pending operations.

Overall, Design v1 considers a single threaded client and server. In the later sections, we will show how we can support multi-threaded clients and servers in uFS.

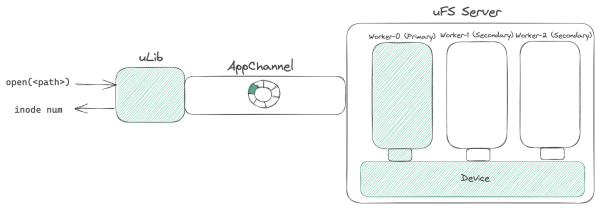

Fig. 2. Making uFS idempotent

B. Design v2: Adding durability

The current design lacks a solution for unpersisted writes on server crashes. To address this, client-side retry of operations needs to be implemented. However, this requires the client library to know when asynchronous operations complete. The previous approach of periodic flushing to disk may not be enough, as metadata related to inode and data bitmaps may have changed during a write. In such cases, a metadata flush is also required, which occurs when the server gracefully exits or is flushed explicitly via `fdatasync()`/`fsync()`/`syncall()` calls.

1) Overview: To notify the client library about flushes and track request completion, we propose to send notifications from the server to the client library. In this section, we discuss the design decisions for identifying flushed requests, maintaining pending request states by the client library, and the server, and how notifications will work. We will only focus on create and write/pwrite commands to keep it simple.

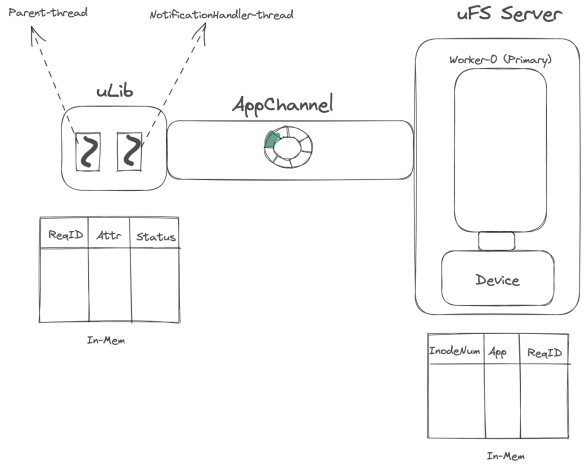

Fig. 3. Durability Notification Handling design

2) Identifying Flushed Requests: We will generate a monotonically increasing request ID on the client library, which we will use in calls made to the server. The client library will maintain a pending request table, which will contain the request ID as the key, request attributes, and the request's status (`FS_PENDING_STATUS`, `FS_SPECULATIVE_STATUS`, or `FS_COMPLETION_STATUS`). The server will also maintain a similar pending request table. When the server flushes requests, it knows which requests were flushed and mentions their request IDs in the notification to the client library. The client library can then update the state of these requests in its pending request table.

3) Pending state maintained by the client library: The client library will maintain a pending request table with the request ID, request attributes, and the request's status. Initially, all requests added to the table will be marked as `FS_PENDING_STATUS`. On notifications, the status will change to either `FS_SPECULATIVE_STATUS` or `FS_COMPLETION_STATUS`. We require these statuses because periodic data buffer flushes do not guarantee that the server will retrieve the data until that point, as metadata changes may not have been flushed. We mark the request as `FS_SPECULATIVE_STATUS` until metadata has flushed, after which we mark it as `FS_COMPLETION_STATUS`. When the request reaches `FS_COMPLETION_STATUS`, we can clear it from the request table. We can maintain the parameters used to make the create and write/pwrite calls but may exclude the buffer for writes to optimize storage.

Additionally, for `fs_allocated_xxx` writes, that use `fs_malloc()` (recall Section II-B), we copy the data buffer to a separate data structure, so that on retries, we can retry the request by coping back the data in the data buffer.

Aside: created a utility command `view_pending_ops`.

4) Pending state maintained by the server: The server will maintain a pending operation table that contains the inode number as the primary key, the request ID as the secondary key, type of request (create/write), and the application's PID that made the request. The inode number as the primary key is useful to move the pending operation state when the inode is reassigned to a different server worker. The PID is useful as the notification can be placed in the application's ring buffer when a flush is made.

5) How do notifications function?: When the server performs a flush (metadata/data), it sends a message for each request ID in the pending operation state to the client library's ring buffer. The server knows the request ID since it maintains the state of all the pending operations. After all the request IDs are flushed, the server sends this information to each client library's ring buffer as a message when the flush completes.

Currently, we use one message per request ID. However, this may be inefficient if the dirty ratio is high (making many requests pending). To improve efficiency, we can batch seven request IDs per message, since the message must fit within a cache line of 64 B. Each request ID is eight bytes (`uint64_t`), and we require one byte to fit the message status (`shmipc_STATUS_NOTIFY_CLIENT`) and one byte for the notification type (`FS_SPECULATIVE` or `FS_COMPLETION`). Therefore, we can fit only seven request IDs (8 bytes * 7 + 2 bytes $<$ 64 bytes) in a message.

The client library has a dedicated thread that polls the ring buffer for messages from the server with status `shmipc_STATUS_NOTIFY_CLIENT`. If the thread finds a message, it handles it based on the type (i.e., `FS_SPECULATIVE` or `FS_COMPLETION`) and updates the status of the corresponding request in the pending request table to the specified type.

Aside: created a utility command `poll_pending_ops`.

6) What happens when the server crashes?: In the event of a server crash with pending operations, the client library has nearly enough information to retry the operations when the server comes back online. The first step is to detect the server crash. One way is to set a timeout of x seconds (as discussed in v1) for the client library to receive a response after placing a request in the buffer. If the client library does not receive a response within this time frame, it can declare that the server

has crashed.The client library can then manually ping the server until it responds with a successful response. If the server does not respond, the client library can invoke the retry command to retry all operations in the pending request table. For create requests, the operation can be retried without any additional information from the client library. For write operations, however, the client library must provide the data again.

Alternatively, the client library can try the operation n times. If there is no response after x seconds, it can declare that the server has crashed. The client library can then automatically ping the server n times before giving up. If the client library gives up, it is up to the client to determine if the service is back up. The client library can automatically run the retry module and request write buffers from the client when necessary.

C. Design v3: Improving durability

Version 2 of the durability design had a flawed approach to handling retries for non-persisted data. It did not store data associated with pending writes, and did not automatically retry pending operations. Instead, it relied on the user of the client to provide this information. In this version, we aim to address those issues.

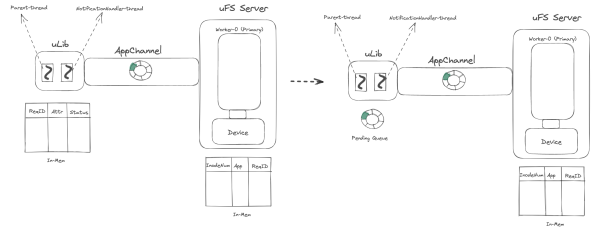

Fig. 4. An improved durability design

1) Client State: To handle updates to the data or metadata of the file system, we have created a separate ring buffer to store operations that can be retried in case the server crashes before flushing these changes. The client now maintains an additional ring buffer along with a mapping of request IDs to ring buffer indexes. Certain requests can span multiple ring indexes, such as write requests, and this mapping is needed to recreate the original request.

In addition, the client now has a dedicated notification handler thread that wakes up every 5 seconds to process any notifications sent by the server on flush.

Furthermore, any operations that cannot be completed, such as a read request when the server is unavailable, are also added to the ring buffer.

2) Server State: The server state remains unchanged from v2. During its periodic flush, the server flushes metadata (by doing an internal sync call) and sends notifications to the client for all requests that update the metadata and data. Therefore, there is no longer a need for `FS_SPECULATIVE_STATUS`. Additionally, when the server exits gracefully, it flushes both metadata and data.

3) Retrying Pending Operations: The client now invokes the retry module to retry all pending requests in increasing order of request ID. This ensures that the requests are processed in the order they were initially triggered to the server.

4) Background Flushes: The pending queue has 64 slots. Once this limit is reached, we invoke `fsync()` to flush any pending requests.

D. Design v4: Improving server crash detection

Previous versions (v1-3) of the server detection mechanism used a simple approach to detect server crashes. However, since the uFS server is a process, we can improve the server crash detection mechanism by checking if the server process is alive.

To achieve this, we can modify the server to send its process id (pid) to the client in the ring buffer every time the server starts up. Whenever the server crashes and restarts, the pid would have changed.

Furthermore, we can use the kill signal to check if the pid is still alive. With this information, we can update the existing server crash detection mechanism. When the client sends a request to the server, it can check if the pid is still alive in the polling sequence. If the kill signal detects that the pid is not alive, we can infer that the server has crashed.

It is important to note that this approach only detects server crashes when the client is actively sending requests to the server. If the server crashes while the client is idle, the client would only detect that the server is up when it sends its next request.

E. Design v5: Optimizations

1) Improving Server Detection: In previous versions, the client used the kill signal to check if the server process was alive. This can be an expensive system call, so we can optimize this by setting a timeout for each request in the polling sequence. If the request status does not change to `READY_FOR_CLIENT` and the timeout is triggered, we can then check if the server process is still alive using the kill signal. This reduces the number of times the kill signal is called.

Additionally, to detect server crashes when the client is idle, we can create a background thread that periodically checks if the server process is alive.

2) Minimizing Data Copy for fs_allocated_xxx: Previously, we copied the data buffer to a separate data structure to be retried in case of a server crash. We can avoid this data copy by keeping a copy of the data pointer in the separate data structure instead. The malloc'd buffer that holds the data is a shared memory space between the client and server, and its memory management is done by the client, ensuring that the data persists even if the server crashes. On retries, we can refer to the data pointer and access the data to resend it. However, with this change, the client cannot free the malloc'd space until the server performs the operation. If none of the writes were flushed and the space is still used, the client must explicitly call fsync to free up the space and proceed with subsequent writes.

3) Notification Handling: In previous versions, a dedicated notification thread was used to handle messages sent from the server to the client. We can optimize this by relying on lazy handling of notification messages. When the client tries to allocate a new slot or request and lands on a slot with a notification message, it clears up the pending state for that request and uses that slot to place the new request. Notification messages are sent by the server to the client in the same ring buffer, with one notification message per request that was flushed.

4) Batching Notification Messages: To further optimize notification handling, we can batch multiple request IDs in a single notification message in the data section of the request. This allows the ring buffer to be used for more incoming requests from the client and simplifies the time it takes for the server to send notifications, as it no longer has to find multiple slots for each request that was flushed.

F. Design v6: Multi-threaded clients

Previously, the uFS was designed to be retriable with a single-threaded client and server. In order to support a multi-threaded client, we need to allow for concurrent generation of request IDs. However, adding locks around the request ID generation would lead to serialization bottlenecks. Instead, we have implemented a lock-free request ID generation scheme. Each thread is assigned a different base to generate request IDs from, and each thread maintains its own monotonically increasing counter to ensure that requests are generated in a thread-safe manner without locks.

Furthermore, to enable concurrent updates to data structures that store pending state information, we have added locks to ensure consistency and prevent data loss.

G. Design v7: Multithreaded server

While we have made progress in making the uFS client retriable with multi-threaded clients, the final challenge is to support multi-threaded servers. As a performant multi-core file system, uFS utilizes multi-threaded workers that are pinned to separate cores. These workers operate in a similar mechanism to GFS, where one thread acts as the primary worker for path-based requests (e.g. open), while the rest of the workers handle other operations to balance the workload. A load balancer is used to distribute the load across the workers by reassigning inodes to workers to proceed with each operation.

To support client retries, we have updated the server to maintain a pending state. However, when reassigning an inode to a different worker, we need to transfer this pending state to the new worker to ensure that the worker can notify the client when it is flushed by the server. Therefore, in the process of reassigning the inode, we also pass the pending state to the new worker. By doing so, we ensure that the uFS server is able to support multi-threading and handle client retries efficiently.

H. Summary

Here's a summary of versions 1-7:

- Version 1: The initial version of uFS has limitations in terms of client retriability, where client retries can result in inconsistencies.
- Version 2: To address the client retry issue, the uFS design introduces a new feature that allows clients to be aware of the server's durability. This new feature requires the server to maintain a pending state for each client request.
- Version 3: To improve the server's ability to handle client retries, the uFS design optimizes the server detection process by declaring a timeout for the request in the polling sequence and uses the ring buffer data structure to hold pending operations.
- Version 4: To improve server crash detection.
- Version 5: The design eliminates the need for a dedicated notification thread by relying on lazy handling of notification messages from the server. It also batches multiple request IDs in a single request to simplify the server's task of sending notifications. The design further improves the efficiency of the file system by minimizing the data copy for `fs_allocated_xxx`.
- Version 6: To support multithreaded clients, the uFS design introduces a lock-free request ID generation process that enables each thread to generate a new request monotonically increasing with respect to that thread.
- Version 7: Finally, to support multi-threaded servers, the uFS design uses a load balancer to distribute the load across workers and reassigns inode to the workers, ensuring that the pending state is transferred to the reassigned worker so that it can notify the client when it is flushed by the server.

IV. Experimentation and Analysis

We use ClouldLab to run all experiments with 24 physical CPU cores on a Linux machine with hardware type c6525-100g.

A. Server crashes

1) Copy workload with server crash: In this experiment, we demonstrate how our uFS system is able to successfully complete copy operations even in the face of server crashes with a single threaded client and server configuration. To do so, we set up a simple copy workload using a 2-level directory structure containing files and directories.

To test the resilience of our system, we use a system intercept library [4] to redirect the calls made to copy the directory to our uFS server. We then intentionally crash the server just before it responds to the client at every system call, and measure the total time it takes to complete the request.

The source directory structure is as follows:

- FSPsrc (ino=13)
  - FSPdirl1_0 (ino=14)
    - FSPf0 (ino=18, size=100KB)
    - FSPf1 (ino=19, size=200KB)
  - FSPdirl1_1 (ino=15)
    - FSPdirl2_0 (ino=16)
      - FSPf3 (ino=21, size=110KB)
    - FSPdirl2_1 (ino=17)
    - FSPf2 (ino=20, size=210KB)

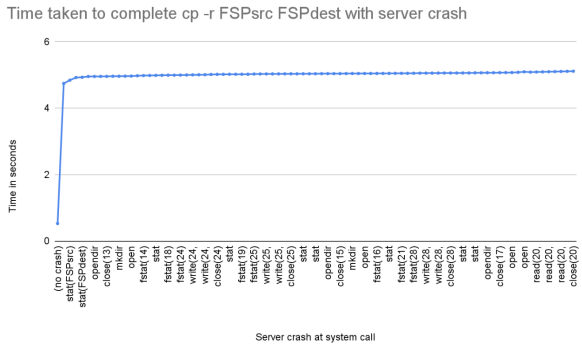

Fig. 5. Copy workload

Overall, our results (Fig. 5) show that the server takes approximately 5 seconds to restart after a crash, while completing the workload without crashes takes less than 1 second. As a result, it is difficult to see the effects of crashing the server at later system calls, as the server's wake-up time dominates the overall completion time. However, we do observe that the server takes slightly longer to complete the request later on in the workload sequence, as the client has to retry multiple requests.

It is worth noting that we did not run the checkpointer before restarting the server. Therefore, the

5-second start-up time is solely for the purpose of turning the server back on and does not account for the durability aspect of writes. Additionally, our readings were taken using version 3 of the system.

Finally, it is interesting to note that our workload includes the ”mkdir” command, which we have made idempotent and thus retriable in the event of a server crash.

B. Sanity check

The objective of this experiment was to verify the effectiveness of our system's retry capabilities in handling multiple client (shown) and server (not shown) threads on Design v7. Results from the copy workload demonstrated that the total time is dominated by server start-up time. We also observed (Fig. 6) that recovering from the middle and end takes longer compared to recovering from the beginning, as a larger number of operations need to be retried. However, the difference between the two is not significant due to the background flush, which helps to limit the client retry time in the event of repeated server crashes. Overall, these findings confirm the reliability of our system in retrying operations in a multi-threaded environment.

Fig. 6. Server crash with multithreaded clients

C. Microbenchmarks

In order to evaluate the impact of client retry support in the absence of server crashes, we utilized several of File Bench's microbenchmarks [5]. These benchmarks were performed using a single client application and a single server/worker. Our workloads included sequential and random reads, sequential and random writes with fsync, create, mkdir, and stat operations. For example, the sequential read microbenchmark entailed single-threaded sequential reads (1KB I/Os) on a 1MB file, while the create microbenchmark involved creating a file and writing 1MB to it. The stat microbenchmark called stat on 50 files (with the file structure the same as the createfiles microbenchmark). Overall, these microbenchmarks allowed us to gain insight into the performance overhead caused by client retry support in the absence of server crashes. This was performed on version 4.

Here are the details for each microbenchmark workload:

- seqread: This microbenchmark involves performing single-threaded sequential reads of 1KB I/Os on a 1MB file.
- rread: This microbenchmark involves performing single-threaded random reads of 1KB I/Os on a 1MB file. The benchmark stops after 1MB has been read.
- seqwrite: This microbenchmark involves performing single-threaded sequential writes of 1KB I/Os to an empty file.
- seqwritesync: This microbenchmark involves performing single-threaded sequential writes of 1KB I/Os to an empty file and calling fsync after every write call.
- rwrite: This microbenchmark involves performing single-threaded random writes of 1KB I/Os on a 1MB file. The benchmark stops after 1MB has been written.
- rwritesync: This microbenchmark involves performing single-threaded random writes of 1KB I/Os to a 1MB file. The benchmark stops after 1MB has been written and calls fsync after every write call.
- seqwriterand: This microbenchmark involves performing single-threaded appends of random-sized I/Os in the range of 1B to 8KB to an empty file. The benchmark stops after 1MB has been written.
- seqwriterandsync: This microbenchmark involves performing single-threaded appends of random-sized I/Os in the range of 1B to 8KB to an empty file. The benchmark stops after 1MB has been written and calls fsync after every 100 write calls (approximately 2 times).
- create: This microbenchmark involves creat-

ing a file and writing 1MB to it.

- createfiles: This microbenchmark involves creating 50 files (10 files per directory) and writing 1MB to each file.
- stat: This microbenchmark involves calling stat on 50 files (with a file structure identical to createfiles).

We see (Fig. 7) that the client-retry-support performs 100x worse than no-client-retry-support, which is our baseline (the unmodified uFS). This is why we introduced v5.

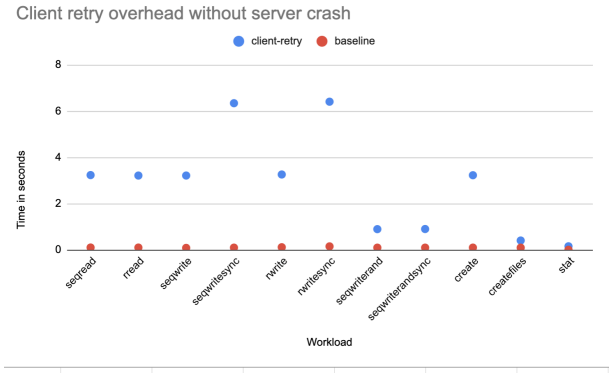

Fig. 7. Microbenchmark on non-optimized version

After running the microbenchmarks on version v5 (Fig. 8), we observed that read-related operations showed similar performance to the baseline, whereas write-related operations took longer. The reason for this is that we have to carry out bookkeeping for pending states and background flushing when the pending queue becomes full.

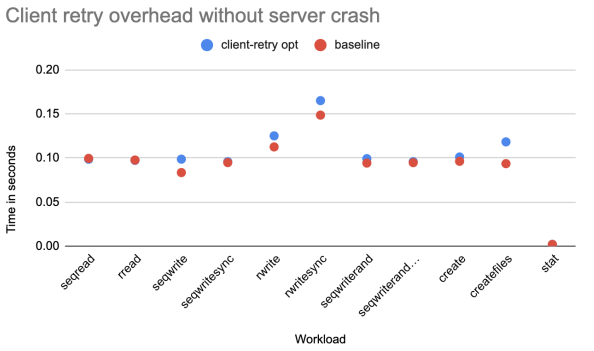

Fig. 8. Microbenchmark on optimized version

D. Throughput

We run longer workloads to acquire the throughput of the client-retries version of uFS. We experiment with sequential and random reads and writes for files up to 1GB. All experiments are run using v7 of the system.

1) Reads: For the sequential and random read workload, each client thread performed the following operations: opened the file system, read 4KB at a time on 2 files of size 100MB using `fs_allocated_read()`, closed the file system. Each client thread completed approximately 50K operations, and the experiment was conducted with client and server threads ranging from 1 to 5. Fig. 9-12 show results for sequential reads, while Fig. 14, 20-23 show results for random reads.

Fig. 9. Sequential Read Throughput with 1 server

Fig. 10. Sequential Read Throughput with 2 servers

As we maintain a constant number of servers, we can observe an increase in throughput with an increase in the number of client threads for both the client-retries (our system) and the baseline (unmodified uFS). This observation is evident for both sequential and random reads. However, there is a slight difference in throughput between the two systems, with client-retries having slightly lower

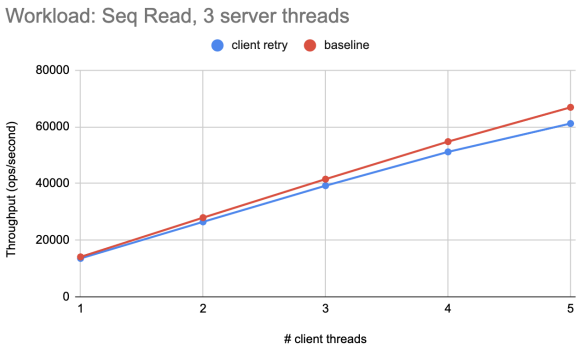

Fig. 11. Sequential Read Throughput with 3 servers

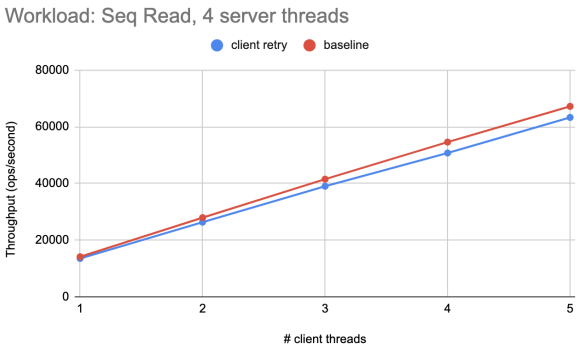

Fig. 12. Sequential Read Throughput with 4 servers

Fig. 13. Sequential Read Throughput with 5 servers

Fig. 14. Random Read Throughput with 1 server

throughput due to extra bookkeeping.

Moving on, we investigate the effect of increasing the number of servers on throughput. Surprisingly, we observe that the throughput does not change significantly. Even with five servers, we observe that 400 operations are completed faster than with one server. Upon examining the load balancing mechanism in uFS, we found that load statistics are collected by a background thread at least every one second, whereas the sequential and random read workload lasts between one to five seconds. As a result, we do not observe any significant differences in throughput with an increase in the number of servers. Henceforth, we show the performance of servers 2-5 in the Appendix.

2) Read and Writes: For the sequential and random read with sequential and random overwrite workload, each client thread performed the following operations: opened the file system, read 4KB at a time on 2 files of size 100MB using `fs_allocated_read()`, wrote 4KB at a time on 2 files of size 100MB using `fs_allocated_write()`, and closed the file system. The experiment was conducted with and without explicit sync at every write call, and each client thread completed approximately 101K (client retries) and 100K baseline operations without explicit sync and 150K operations with explicit sync. The experiment was conducted with client and server threads ranging from 1 to 5. Fig. 15, 24-27 show results for sequential read and write, Fig. 16, 28-31 show results for sequential read and write with sync, Fig. 17, 32-35 show results for random read and write, and finally, Fig. 18, 36-29 show results for random read and write with sync.

When keeping the number of servers constant, we noticed that the throughput, similar to the read workloads, such as sequential and random reads, increased as we added more client threads. However, we also observed a significant difference in throughput between client-retries and the baseline when no explicit sync was used. This can be attributed to the fact that the baseline does not perform any background syncs, while the client-

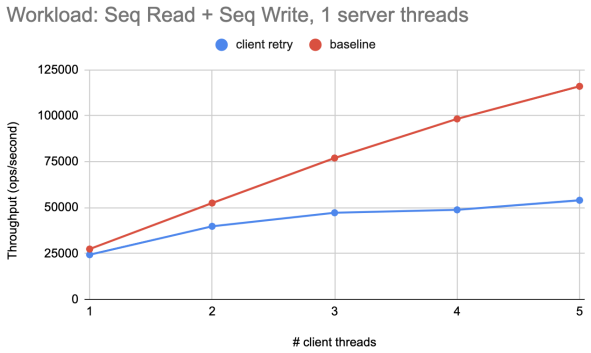

Fig. 15. Sequential Read and Write Throughput with 1 server

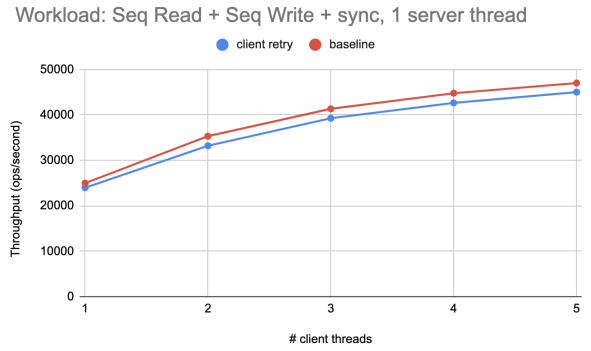

Fig. 16. Sequential Read and Write Sync with 1 server

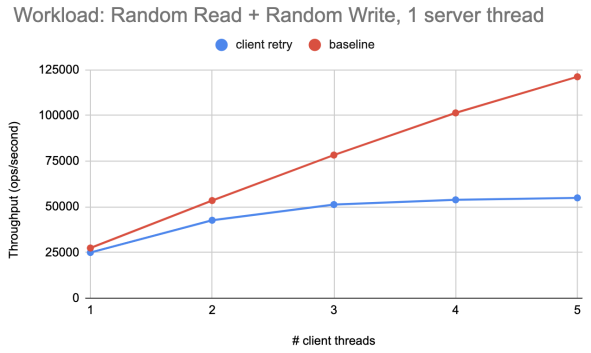

Fig. 17. Random Read and Write Throughput with 1 server

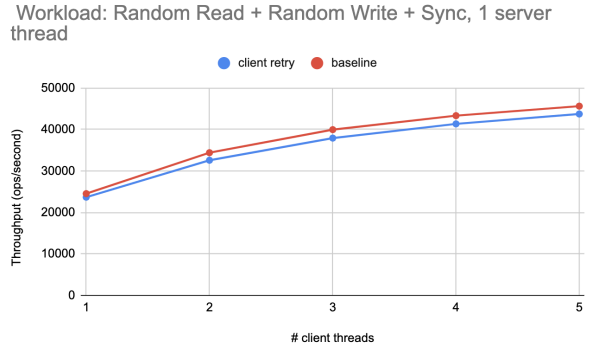

Fig. 18. Random Read and Write Sync Throughput with 1 server

retries system needs to do so due to the limited slots in the pending queue (Refer IV-E). By forcing syncs for both client-retry and baseline after every write, we were able to achieve similar throughput. As for the number of server threads, we see very minor improvements by using more than 1 server. It is possible that the performance of a single server is not saturated by this workload. Another aspect wrt multithreaded server and client retries is that most of the workload is a synchronous operation (fsync).

E. Per write performance

This workload involved each client thread performing `fs_allocated_write()` 4KB at a time for 64 times to request a wraparound of the size of the ring buffer. This experiment was conducted with a single client thread and a single server thread, using version 7 of the system. Results are shown in Fig. 19.

From the results, we can see that the client-retries and baseline perform similarly until the point where the client-retries need to perform a background flush due to the size of the pending queue. These observations were also made using version 7 of the system.

V. Conclusion

In conclusion, our work aimed to enhance the functionality of the uFS, a crash-consistent file system in user space, by introducing client retry capabilities. Our objective was to make the file system more robust and resilient to potential server failures. To achieve this, we made the API idempotent, which meant that it could be safely retried without changing the result of the previous operation. Additionally, we designed the client to keep track of all pending operations and operations that were not persisted by the server. If the pending requests hit a limit, the client forces flush requests to ensure that the file system remains crash-consistent.

It is worth noting that the implementation of the client retry capabilities required the client to under-

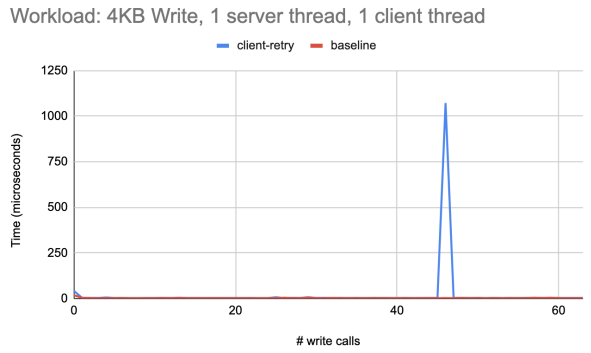

Fig. 19. Time to complete writes

take a lot of heavy lifting. However, we believe that the benefits of increased resilience and improved crash-consistency make the effort worthwhile. Furthermore, the performance overhead caused by the client retry capabilities is generally acceptable, given the enhanced functionality and robustness that they provide.

To evaluate the effectiveness of our client retry capabilities, we subjected the file system to various workloads, including a simple webserver workload with a multi-threaded client and a copy workload on a single-threaded client. In both cases, we showed that the client library was able to handle server crashes and continue operations seamlessly. These results demonstrate the value of our approach in enhancing the reliability and resilience of the file system.

Finally, we demonstrated that adding client retry capabilities to the file system comes at a cost, primarily in terms of write performance. We observed that the periodic flushes required to hold a limited number of pending requests in the queue significantly impacted the performance of write operations. This observation highlights the importance of carefully balancing the trade-offs between performance and reliability when designing file systems with client retry capabilities.

VI. Acknowledgments

I would like to express my sincere appreciation to Professor Andrea for providing me with the opportunity to undertake this project. Her guidance and support throughout this endeavor have been invaluable, and I am deeply grateful for her mentorship. Additionally, I would like to thank Jing for their invaluable contributions to my understanding of the uFS system. Their support and assistance have been instrumental in helping me achieve my goals and complete this project successfully.

VII. Appendix

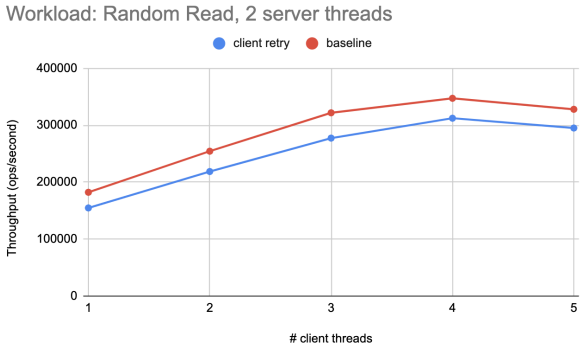

Fig. 20. Random Read Throughput with 2 servers

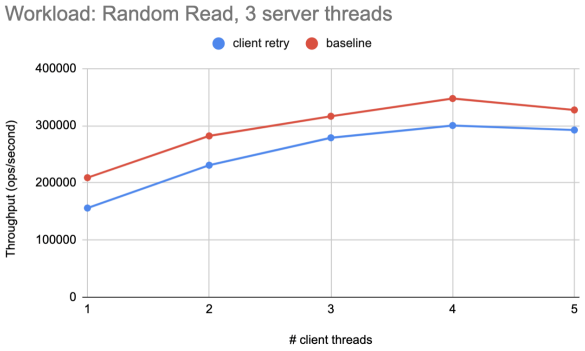

Fig. 21. Random Read Throughput with 3 servers

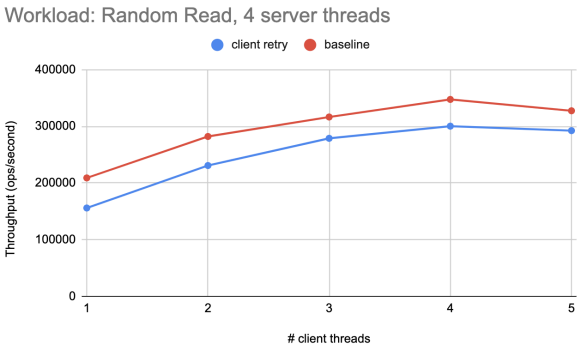

Fig. 22. Random Read Throughput with 4 servers

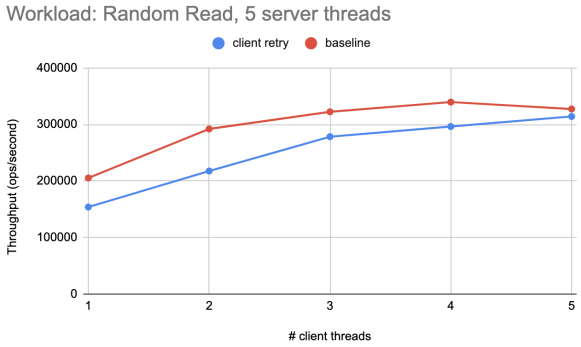

Fig. 23. Random Read Throughput with 5 servers

References

[1] Original uFS codebase, https://github.com/WiscADSL/uFS
[2] Client retries over uFS codebase, https://github.com/benitakbritto/uFS
[3] Independent Study progress log, https://docs.google.com/document/d/11E4DnFbFnhwPV0Ft41B7Bkh52UY2
[4] Syscall Intercept codebase, https://github.com/jliu9/syscall_intercept
[5] FileBench, https://github.com/filebench/filebench
[6] Jing Liu et. al, Scale and Performance in a Filesystem Semi-Microkernel, https://research.cs.wisc.edu/adsl/Publications/ufs-sosp21.pdf
[7] NFSv2, https://en.wikipedia.org/wiki/Network_File_System
[8] uFS API, https://github.com/WiscADSL/uFS/blob/main/cfs/include/fsapi.h
[9] Sanjay Ghemawat et. al, The Google File System, https://static.googleusercontent.com/media/research.google.com/en//archive/g sosp2003.pdf

Fig. 24. Sequential Read and Write Throughput with 2 servers

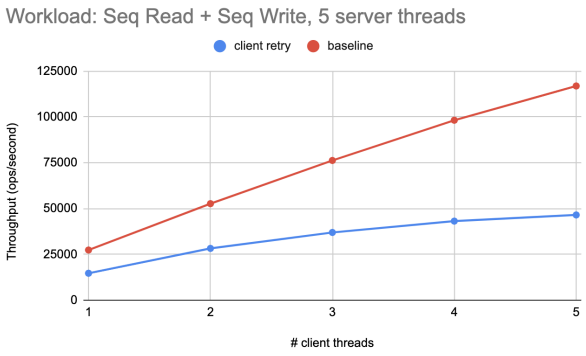

Fig. 27. Sequential Read and Write Throughput with 5 servers

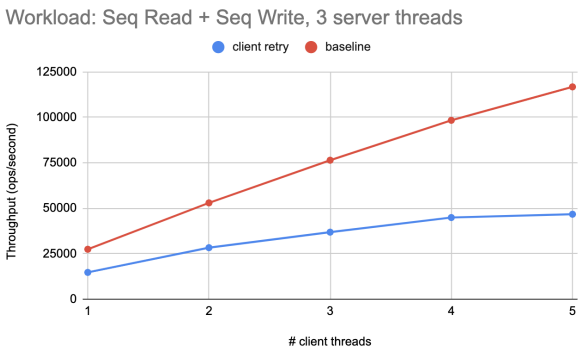

Fig. 25. Sequential Read and Write Throughput with 3 servers

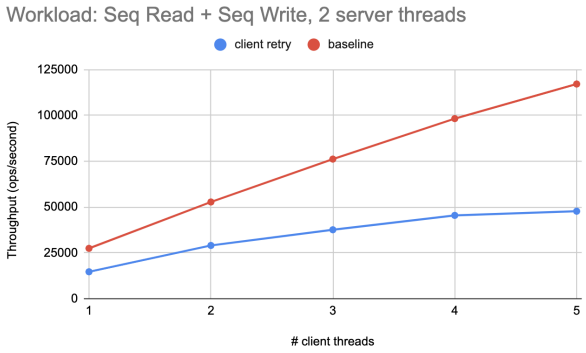

Fig. 28. Sequential Read and Write Throughput with 2 servers

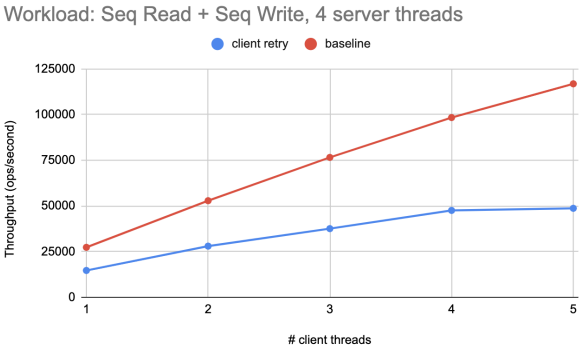

Fig. 26. Sequential Read and Write Throughput with 4 servers

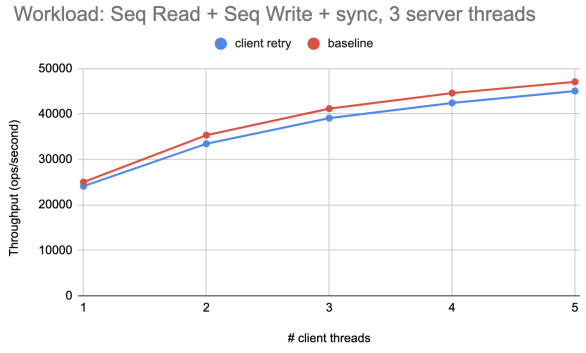

Fig. 29. Sequential Read and Write Sync with 3 server

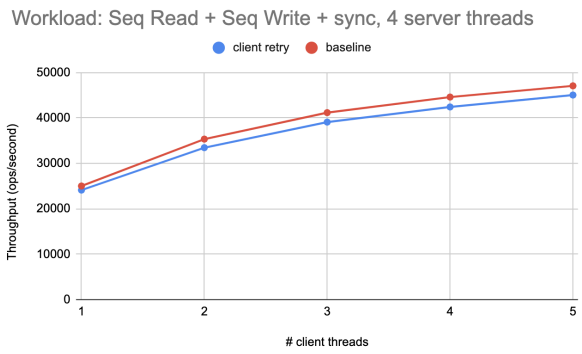

Fig. 30. Sequential Read and Write Sync with 4 server

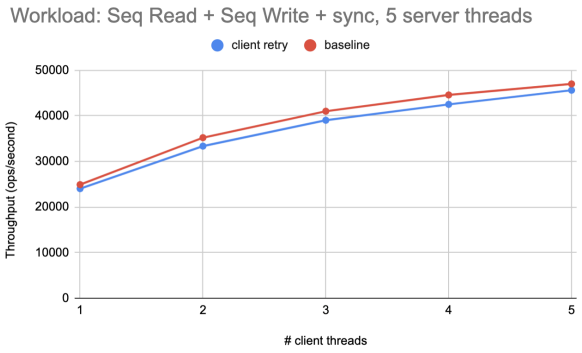

Fig. 31. Sequential Read and Write Sync with 5 server

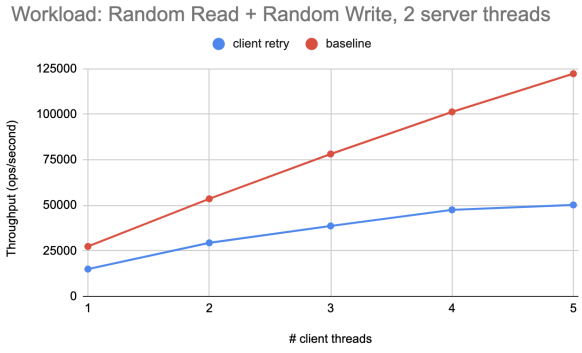

Fig. 32. Random Read and Write Throughput with 2 servers

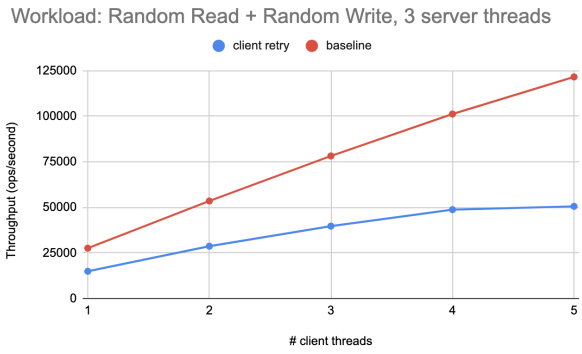

Fig. 33. Random Read and Write Throughput with 3 servers

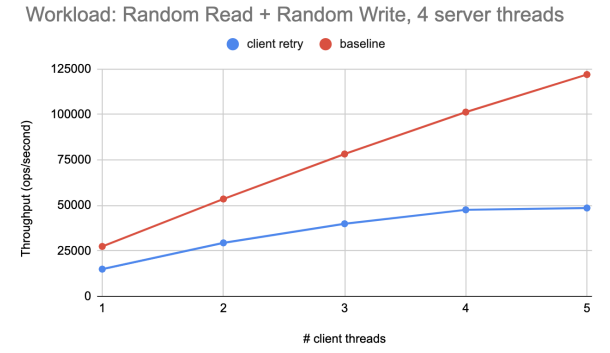

Fig. 34. Random Read and Write Throughput with 4 servers

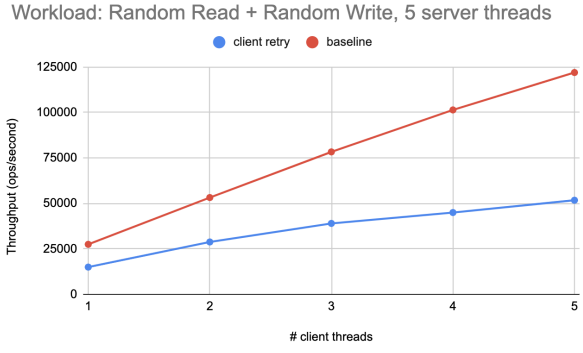

Fig. 35. Random Read and Write Throughput with 5 servers

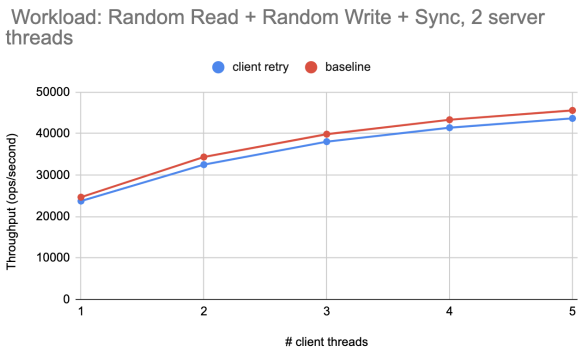

Fig. 36. Random Read and Write Sync Throughput with 2 servers

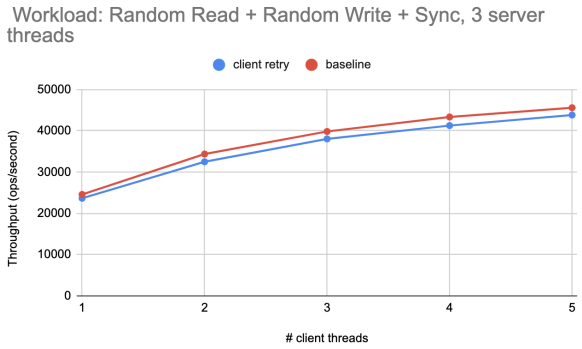

Fig. 37. Random Read and Write Sync Throughput with 3 servers

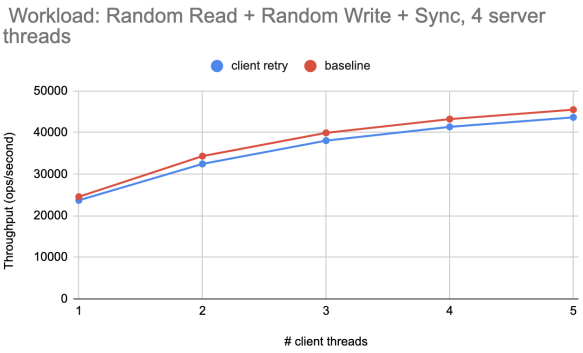

Fig. 38. Random Read and Write Sync Throughput with 4 servers

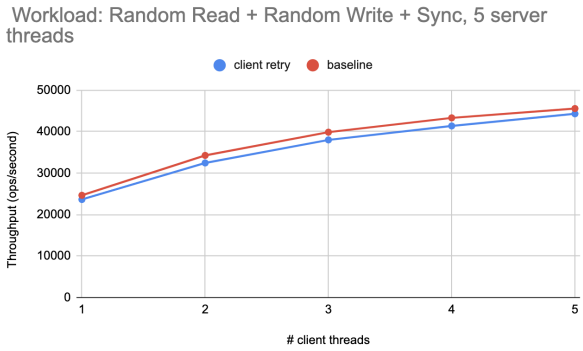

Fig. 39. Random Read and Write Sync Throughput with 5 servers