MANTECH Publications

Recent Trends in Computer Science and Software Technology
Volume 8, Issue 3, September-December 2023

Optimizing Video File Distribution: A Systematic Approach with the Odds Algorithm

Praveen Rai[1], Vijneesh Lokhande[2]

Assistant Professor[1], Student[2]

Department of CSE

Sri Rammurti Smarak College of Engineering Technology & Research

Corresponding Author's Email id: *vijneeshlokhande255@gmail.com*

Abstract

In the ever-expanding digital landscape, the efficient distribution of video files has become a critical challenge. This paper introduces a systematic approach to optimize video file distribution through the implementation of the Odds Algorithm. The Odds Algorithm, a probabilistic method, is employed to enhance the allocation and delivery of video content across diverse networks. The systematic approach outlined in this paper aims to address the complexities of video file distribution, considering factors such as bandwidth constraints, network latency, and user demand. By leveraging the Odds Algorithm, our proposed system optimizes the distribution process, resulting in improved resource utilization and enhanced user experience. This research contributes to the advancement of video distribution systems, offering a practical and effective solution to the challenges associated with large-scale content dissemination.

Keywords: *Video File Distribution, Odds Algorithm, Optimization, Content Delivery, Network Efficiency, Bandwidth Management, Systematic Approach, Digital Content, Probabilistic Methods, User Experience.*

INTRODUCTION

ODDS leverage a distributed file system (DFS) to provide scalable data access for scientific analysis. Through exploiting the information of underlying data distribution in DFS, ODDS employs a novel data-locality scheduler to transform a compute-centric mapping into a data-centric one and enables each computational process to access the needed data from a local or nearby storage node.

ODDS are suitable for parallel applications with dynamic process-to-data scheduling and for applications with the static process-to-data assignment. We have designed and implemented the Video File Distribution System to meet the rapidly growing demands data processing needs. It shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application work- loads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, files are huge by traditional standards. Multi-GB files are common. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of ap- proximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the fo- cus of performance optimization and atomicity guarantees.

DESIGN OVERVIEW

2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that off er both challenges and op- portunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.

- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small ran- dom read typically reads a few KBs at some arbitrary off set. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.

- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary posi- tions in a file are supported but do not have to be effcient.

2.2Interface

The concept provides a familiar file system interface. Files are organized hierarchically in directories and identified by path- names. We support the usual operations to create, delete, open, close, read, and write files.

2.3 Architecture

The cluster consists of a single master and multiple chunk servers and is accessed by multiple clients, as shown in Figure 1 It is easy to run both a chunkserver and a client on the

same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into optimized chunks. Each chunk is identified by an immutable and unique optimized chunk handle assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as data files andread or write chunk data specified by a chunk handle and byte range

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state.

Tclient code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the applica- tion. Clients interact with the master for metadata opera-tions, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placementand replication decisions using global knowledge. However,

we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with refer- ence to Figure 1. First, using the fixed chunk size, the client translates the file name and byte off set specified by the ap- plication into a chunk index within the file. Then, it sends the master a request containing the file name and chunkindex. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key. The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the informa- tion for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost. **See Figure: 1**

2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file sys- tem block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our work- loads because applications mostly read and write large files sequentially.

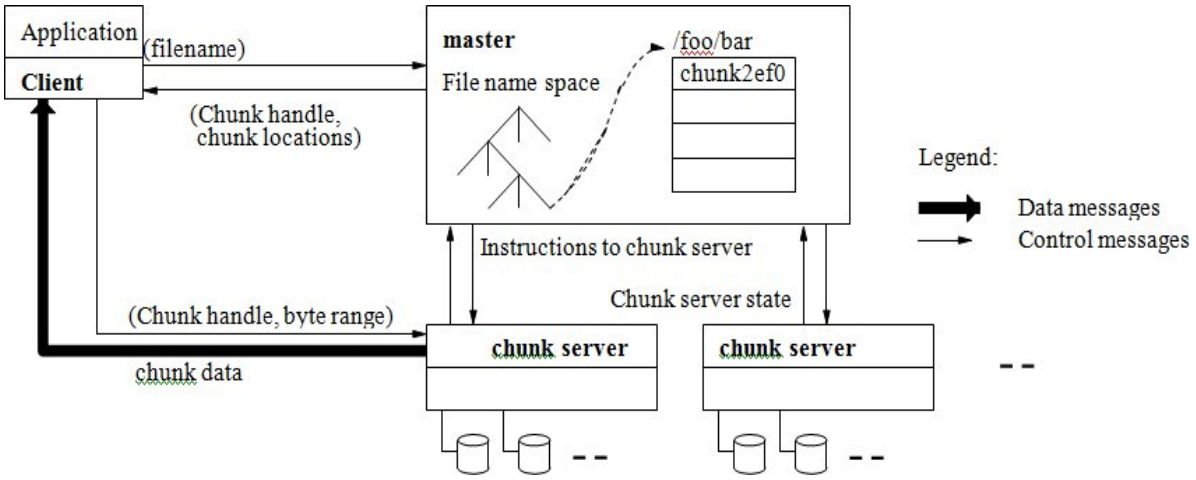

Figure1: Architecture

Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunksize, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when Video File Distribution System was first used by a batch-queue system: an executable was written to Video File Distribution System as a single-chunk file and then started on hundreds of ma- chines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous re- quests. We fixed this problem by storing such executables with a higher replication factor and

by making the batch- queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (names- paces and file-to-chunk mapping) are also kept persistent by logging mutations to an operation log stored on the mas- ter's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location informa- tion persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

2.6.1 in-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver fail- ures, and chunk migration to balance load and disk space usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The mas- ter maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Sim- ilarly, the file namespace data typically requires less then64 bytes per file because it stores file names compactly us- ing prefix compression.

If necessary to support even larger file systems, the costof adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

2.6.2 Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with reg- ular HeartBeat messages.We initially attempted to keep chunk location information persistently at the master, but we decided that it was muchsimpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.Another way to understand this design decision is to real- ize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to Video File Distribution System. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent op- erations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created. Since the operation log is critical, we must store it reli- ably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client opera- tion only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loadingthe latest checkpoint from local disk and replaying only the limited number of log records after that.

The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without ex- tra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the mas- ter's internal state is structured in such a way that a new checkpoint can be created without delaying incoming muta-tions. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint andsubsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

Table 1: File Region State after Mutation

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent* successes
Concurrent but *undefined*	*consistent*	
Failure	*inconsistent*	

3. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Com- ponent failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose prob- lems when they inevitably occur.

3.1 Data Integrity

Each chunk server uses check summing to detect corruption of stored data. Given that a Video File Distribution System cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of Video File Distribution System mutations, in particular atomic record append as discussed earlier, does not guar- antee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by main- taining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver.

Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. Video File Distribution System client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the check- sum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corrup- tion in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

3.2 Diagnostic Tools

Extensive and detailed diagnostic logging has helped im- measurably in problem isolation, debugging, and perfor- mance analysis, while incurring only a minimal cost. With- out logs, it is hard to understand transient, non-repeatable interactions between machines. Video File Distribution System servers generate di- agnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records

on different machines, we can reconstruct the en-tire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

4. MEASUREMENTS

In this section we present a few micro-benchmarks to illus- trate the bottlenecks inherent in the architecture and implementation, and also some numbers from real clusters in use.

4.1 Micro-benchmarks

We measured performance on a Video File Distribution System cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 Video File Distribution System server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

4.1.1 Reads

N clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunk servers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buff er cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for N clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The effciency drops from80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

4.1.2 Writes

N clients write simultaneously to N distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s be- cause we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or2.2 MB/s per client), about half the theoretical limit. As inthe case of reads, it becomes more likely that multiple clientswrite concurrently to the same chunkserver as the numberof clients increases. Moreover, collision is more likely for 16writers than for 16 readers because each write involves three different replicas. Writes are slower than we would like. In practice this hasnot been a major problem because even though iincreasesthe latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered bythe system to a large number of clients.

4.2 Real World Clusters

We now examine two clusters in use within that are representative of several others like them. Cluster A is used regularly for research and development by over a hun- dred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single "task" consists of many processes on many machines reading and writing many files simultaneously.

Table 2: Characteristics of two clusters

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

4.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. "Used space"includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

4.2.2 Metadata

The chunkservers in aggregate store tens of GBs of meta- data, mostly the checksums for 64 KB blocks of user data. The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master's memory does not limit the system's capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file own- ership and permissions, mapping from files to chunks, and each chunk's current version. In addition, for each chunk westore the current replica locations and a reference count for implementing copy-on-write. Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to60 seconds – until it has fetched chunk location informationfrom all chunkservers.

4.2.3 Read and Write Rates

Table 3 shows read and write rates for various time pe- riods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of Video File Distribution System) The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load.

Table 3: Performance Metrics for Two GFS Clusters

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network con- figuration can support 750 MB/s, so it was using its re- sources effciently. Cluster B can support peak read rates of1300 MB/s, but its applications were using just 380 MB/s.

4.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of Video File Distribution System, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which con- tained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data struc- tures.

4.2.5 Recovery Time

After a chunkserver fails, some chunks will become under- replicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the im- pact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps).

All chunks were restored in 23.2 min- utes, at an effective replication rate of 440 MB/s. In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660

GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

4.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two Video File Distribution System clusters comparable but not identi- cal to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

4.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include inter-server requests to carry out client requests or internal back- ground activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by Video File Distribution Systemservers. For example, Video File Distribution Systemclient code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise.

Ex- plicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to re- compile and restart thousands of running clients to do so and cumbersome to collect the results from as many ma- chines.One should be careful not to overly generalize from our workload.Mutual influence may also exist between general applications, and file systems, but the effect is likely more pronounced in our case.

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and trans- ferred, rather than the amount requested.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

4.3.2 Chunk server Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (un- der 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.A significant number of reads return no data at all in clus- ter Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Pro- ducers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buff er- ing within the writers. Writers that buff er less data, check- point or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does becauseour production systems, which use cluster Y, are more aggressively tuned for Video File Distribution System.

4.3.3 Appends versus Writes

Record appends are heavily used especially in our pro- duction systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts.

For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ra- tios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two appli- cations with particular buff er size choices.As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This approximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload per se but a consequence of the retry mechanism.

Table 5: Bytes Transferred Breakdown by Opera- tion Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount re- quested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 6: Master Requests Break down by Type(%)

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

1. MasterWorkload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (FindLo- cation) for reads and lease holder information (FindLease-Locker) for data mutations.

Clusters X and Y see significantly diff erent numbers of Delete requests because cluster Y stores production data sets that are regularly regenerated and replaced with newerversions. Some of this diff erence is further hidden in the diff erence in Open requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode "w" in Unix open terminology).FindMatchingFiles is a pattern matching request that sup- ports "ls" and similar file system operations. Unlike otherrequests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global appli- cation state. In contrast, cluster X's applications are under more explicit user control and usually know the names of all needed files in advance.

EXPERIENCES

In the process of building and Deploying Video File Distribution System, we have experienced a variety of issues, some operational and some technicalInitially, Video File Distribution System was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked,but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in

the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of fsync(). Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an mmap() call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing mmap() with pread() at the cost of an extra copy. Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

RELATED WORK

Like other large distributed file systems such as AFS [5], provides a location independent namespace which en- ables data to be moved transparently for load balance or fault tolerance. Unlike AFS, Video File Distribution System spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, Video File Distribution System cur- rently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], Video File Distribution System does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minand rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp to provide high availability with stronger consistency guarantees than our current scheme.We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, Video File Distribution Systemassumes large number of unreliable components and so fault tolerance is central to our design.Video file distributed system most closely resembles the NASD architecture. While the NASD architecture is based on network-attached disk drives, Video File Distribution Systemuses commodity machines as chunkservers,as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, Video File Distribution System implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's and NASD, we do not seek to alter the model of the storage device. We focus on ad- dressing day-to-day data processing needs for complicated distributed systems with existing commodity components.The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queuesin River .

While River uses memory-based queues dis- tributed across machines andcareful data flow control, uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n dis- tributed queues but lacks the fault tolerance that comes with persistent storage, while it only supports m-to-1 queues effciently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

CONCLUSIONS

The Video File Distribution System demonstrates the qualities es- sential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data pro- cessing tasks of a similar magnitudeand cost consciousness. We started by reexamining traditional file system assump- tions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically diff erent points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequen- tially), and both extend and relax the standard file systeminterface to improve the overall system.Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunk server failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently re- pairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involve- ment in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to pri- mary replicas in data mutations. This makes possible a sim- ple, centralized master that does not become a bottleneck. We

believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

Video file distributed system has successfully met our storage needs and is widely used within as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

REFERENCES

1. Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Server less network file systems. In Proceedings of the 15thACMSymposiumon Operating System Principles, pages109 126,Copper Mountain Resort, Colorado, December1995.

2. Remzi H.Arpaci-Dusseau, Eric Anderson, Noah Treuhaft ,DavidE.Culler, Joseph M. Hellerstein, David Patterson, and KathyYelick.ClusterI/Owith River: Making the fast case common. In Proceedings of the Sixth Workshopon Input/ Outputin Parallel and Distributed Systems(IOPADS'99),pages10–22, Atlanta,Georgia,May1999.

3. Luis-Felipe Cabreraand Darrell D.E. Long. Swift: Using distributed disks triping to provide high I/O datarates.ComputerSystems,4(4):405–436,1991.

4. Garth A.Gibson, DavidF.Nagle, KhalilAmiri, Jeff Butler, FayW.Chang, Howard Gobioff, Charles Hardin, ErikRiedel, David Rochberg, and Jim Zelenka.Acost-effective, high-bandwidth storage architecture. In Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems, pages92–103,SanJose,California,October199

5. John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scaleand

performanceinadistributedfilesystem.ACM Transactions on Computer Systems, 6(1):51–81, February1988. InterMezzo.http://www.inter-mezzo.org,2003.

6. Barbara Liskov ,Sanjay Ghemawat, Robert Gruber, PaulJohnson, Liuba Shrira, and Michael Williams. ReplicationintheHarpfilesystem.In13th Symposium on Operating System Principles, pages 226–238,PacificGrove,CA,October1991.Lustre.http://www.lustreorg,2003.

7. David A.Patterson, Garth A.Gibson, and RandyH. Katz.A case for redundant array so fin expensive disks (RAID).InProceedingsofthe1988ACMSIGMOD International Conference on Management of Data, pages109–116,Chicago,Illinois,September1988.

8. Frank Schmuck and Roger Haskin. GPFS: Ashared-disk file system for large computing clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages231–244,Monterey,California,January2002.

9. Steven R. Soltis, Thomas M.Ruwart, and Matthew T.O' Keefe. The Gobal File System. In Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, College Park, Maryland, September1996.

10. Chandramohan A.Thekkath, Timothy Mann, and Edward K.Lee.Frangipani:A scalabled is tribute file system.InProceedingsofthe16thACMSymposium on Operating System Principles, pages224–237, Saint-Malo, France, October 1997.