Towards a Distributed Multi-tier File System for Cluster Computing

(Position Paper)

Herodotos Herodotou
Cyprus University of Technology
Limassol, Cyprus
Email: herodotos.herodotou@cut.ac.cy

***Abstract*—Distributed storage systems running on clusters of commodity hardware are challenged by the ever-growing data storage and I/O demands of modern large-scale data analytics. A promising trend is to exploit the recent improvements in memory, storage media, and network technologies for sustaining high performance at low cost. While recent work explores using memory and SSDs as a cache for local storage or combining local with network-attached storage, no work has ever looked at all layers together in a distributed setting. We present a novel design for a distributed file system that is aware of heterogeneous storage media (e.g., memory, SSDs, HDDs, NAS) with different capacities and performance characteristics. The storage media are explicitly exposed to users and applications, allowing them to choose the distribution and placement of replicas in the cluster based on their own performance and fault tolerance requirements. At the same time, the system offers a variety of pluggable policies for automating data management for increased performance and better cluster utilization. We analyze the new trends and challenges that led to our application- and data-centric design choices, and discuss how those choices inspire new research opportunities for data-intensive processing systems.**

I. Introduction

Commodity machines on compute clusters have seen significant improvements in terms of memory, storage media, and network technologies. Memory capacities are constantly increasing, which lead to the introduction of new in-memory data processing systems, like Spark [1]. On the storage front, flash-based solid state drives (SSDs) offer low access latency and low energy consumption but are still relatively expensive, making hard disk drives (HDDs) the predominant storage media in datacenters today [2]. Finally, network-attached enterprise storage has been coupled with direct-attached storage in cluster environments for improving data management [3]. This heterogeneity of available storage media with different capacities and performance characteristics must be taken into consideration while designing the next generation of distributed storage and processing systems.

At the same time, modern applications exhibit a variety of I/O patterns: batch processing applications (e.g., MapReduce [4]) care about raw sequential throughput, interactive query processing (e.g., via Hive [5]) benefits from lower latency storage media, whereas other applications (e.g., HBase [6]) make use of random I/O patterns. Hence, it is desirable to have a variety of storage media and let each application choose the one that best fits its performance, cost, and durability requirements.

Recent work takes advantage of the increase in memory sizes and utilizes caching or re-computation through lineage for improving local data access in distributed applications [7], [2], [1]. SSDs have also been used recently as the storage layer for distributed systems, such as key-value stores [8] and MapReduce systems [9]. Finally, [10], [3] focus on improving data retrieval from remote enterprise or cloud storage systems to local computing clusters by utilizing on-disk caching at the cluster compute nodes.

Whereas previous work explores using memory and SSDs as a cache for local storage, and local storage as a cache for remote storage, no work has ever looked at *all layers* together in a distributed setting. In this position paper, we argue for the need of—and present a novel design for—a *distributed, multi-tier file system (MTFS)* that utilizes multiple storage media with different performance characteristics for storing data. Our design focuses on two antagonistic system capabilities: *controlability* and *automatability*. On one hand, the heterogeneous storage media are explicitly exposed to users and applications, allowing them to choose the distribution and placement of replicas in the cluster based on their own performance and fault tolerance requirements. On the other hand, MTFS offers a variety of pluggable policies for automating data management with the dual goal of increasing performance throughput while improving cluster utilization.

The key challenge lies in the creation of the appropriate abstractions that simplify and automate data management across storage tiers yet give enough control to users and applications to satisfy their varying requirements. In this way, higher-level processing systems can take advantage of the unique capabilities of MTFS to build *their own automated management features*. At the same time, MTFS offers the de facto features of fault tolerance, scalability, and high availability. Finally, in order to support multitenancy, the system offers security measures and quota mechanisms per storage media to allow for a fair allocation of limited resources (like memory and SSDs) across users.

Our high-level design is inspired by other popular distributed file systems such as GFS [11] and HDFS [12]. We believe this work will open new research directions for distributed systems and, moving forward, we plan to utilize its unique capabilities for improving the functionality of various systems, such as the task scheduling algorithms of MapReduce and Spark, the query processing of Pig and Hive, the workload scheduling of Oozie, and many others.

II. System Architecture

MTFS enables scalable and efficient data storage on compute clusters by utilizing directly-attached HDDs, SSDs, and memory, as well as remote storage (network-attached or cloud storage). It is designed to store and retrieve *files*, whose data will be striped across nodes and replicated for fault tolerance. MTFS employs a *multi-master/slave* architecture similar to HDFS [12], shown in Figure 1.

A. Primary and Backup Masters

A *Primary Master* is responsible for maintaining two metadata collections, the *directory namespace* and the *block locations*. The directory namespace offers a traditional hierarchical file organization as well as typical operations like creating, deleting, and renaming files and directories. The file content is split into large blocks (128MB by default) and each block is independently replicated at multiple Workers. The Master also maintains the mapping of file blocks to Workers per storage media available. In order to scale the name service horizontally, multiple Masters are used to form a *federation* and are independent from each other.

Each Primary Master can have a *Backup Master* for increased fault tolerance and availability. The Backup is responsible for periodically creating and persisting a checkpoint of the namespace metadata so that the system can start from the most recent checkpoint upon a Master's failure. In addition, it maintains an in-memory up-to-date image of the namespace and is standing by to take over in case the Master fails.

B. Workers

The *Workers* are run one per node in the cluster and are responsible for (i) storing and managing the file blocks on the various storage media, (ii) serving read and write requests from the file system's Client, and (iii) performing block creation, deletion, and replication upon instructions from the Masters. Each Worker is configured to use the available storage media in the node it is running on. The same type of storage media with similar I/O characteristics (e.g., SSD devices) across all Workers are logically grouped into a virtual *storage tier* (e.g., the "SSD" tier). If some nodes have different types of SSDs, for example PCIe and SATA SSDs, with different performance characteristics, the system can be configured to use two tiers for them (e.g., "SSD-1" and "SSD-2").

The file blocks can be stored and replicated in one or more tiers, based on requests from the Client or pluggable management policies. For example, consider the cluster in Figure 1 that shows 4 tiers, namely "Memory", "SSD", "HDD", and "Remote". A block may have 3 replicas on the "SSD" tier (on 3 different nodes); or 1 replica on each of the "Memory", "SSD", and "HDD" tier (on 1, 2, or 3 different nodes); or any other combination. Hence, users and applications have tremendous flexibility on how to place and move data in MTFS.

When remote storage is attached to MTFS, applications can use any of the Workers for reading and writing file blocks. Reading a block from remote storage is equivalent to reading a part of that file given an offset and a length. Hence, multiple Workers can read different parts of the same file in parallel. Management policies can also be used for caching data in one of the higher-level tiers for improving future data accesses.

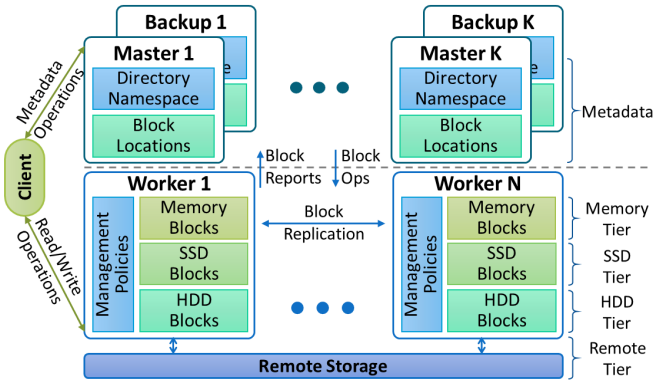

Fig. 1. Multi-tier File System (MTFS) architecture.

C. Clients

A user or application interacts with MTFS through the *Client*, which exposes APIs for all typical file system operations. The management of files with respect to the storage tiers is achieved through a *replication vector* that specifies the number of replicas for each tier. For example, the replication vector V =<"Memory", "SSD", "HDD", "Remote">=$< 1, 0, 2, 0 >$ for a file F indicates that F has 1 replica in the "Memory" tier and 2 replicas in the "HDD" tier. V can be used during F's creation for specifying the desired replication factor per tier for storing F.

A new API allows users to modify the replication vector of a file and achieve various functionalities, including moving a file between tiers (e.g., by changing $< 1, 0, 2, 0 >$ to $< 1, 1, 1, 0 >$), copying a file between tiers (e.g., by changing $< 1, 0, 2, 0 >$ to $< 1, 1, 2, 0 >$), modifying the number of replicas within a tier (e.g., by changing $< 1, 0, 2, 0 >$ to $< 1, 0, 3, 0 >$), and deleting a file from a tier (e.g., by changing $< 1, 0, 2, 0 >$ to $< 0, 0, 2, 0 >$).

Each time V changes, a *network-aware* and *tier-aware* placement policy is invoked for deciding where the addition or deletion of a replica will take place. Finally, the Client exposes both the locations and the tiers of the replicas, allowing applications to make a fully-informed decision for which replica to read from for improving the read I/O performance.

III. Data Operations

The awareness of storage media with different performance characteristics adds a significant level of complexity to the main file operations of the system and creates the need for new placement and retrieval policies. At the same time, the new design offers the potential for using the file system as a multi-level caching service as well as seamlessly integrating MTFS with existing remote storage solutions.

A. Data Placement

An application adds data to MTFS by creating a new file and writing the data to it one block at a time using the Client. Upon a block creation, the Client first contacts the Master and obtains a list of <Worker, Tier> pairs that will host the replicas of that block, based on a *pluggable block placement policy*. Next, the Client organizes a Worker-to-Worker pipeline and sends the data. For example, suppose the block locations

are [$< W_1, M >$, $< W_3, H >$, $< W_6, H >$]. Here, the data is pipelined from the "Memory" tier in Worker W_1 to the "HDD" tier in W_3 to the "HDD" tier in W_6.

Our default block placement policy offers a tradeoff between minimizing the write cost and maximizing data reliability and read I/O performance. The placement decision is made along two axes: the *network topology* and available *storage tiers*. The goal of the network-aware part is to improve fault tolerance by making replicas across racks, while the goal of the tier-aware part is to increase I/O performance.

We use the same network-aware approach as HDFS [12]. When a new block is created, the first replica is placed on the node where the Client is located, the second and third replicas are placed on two different nodes in a different rack, and the rest are placed on random nodes. The second aspect of our policy takes into consideration the presence of multiple storage tiers and selects them in a way that balances performance with storage capacity. Finally, combining the network-aware and tier-aware data placement approaches allows MTFS to place replicas across nodes and racks as well as across tiers.

B. Data Retrieval

When an application reads a file, the Client contacts the Master for the list of <Worker, Tier> pairs that host replicas of the file blocks and then contacts the first Worker directly for transferring the data. The Master returns the list ordered based on a *pluggable data retrieval policy*. The order is determined using the network location of the Client, the network locations of the Workers, as well as the storage tiers that host the replicas. On one hand, the Client should read the replica from its nearest Worker in order to improve the read performance and reduce network traffic. On the other hand, the Client should access the replica from the fastest tier for improving the I/O latency. However, the nearest replica may be on a slower tier whereas a replica on a faster tier is on a more distant node. The policy must be aware of the tradeoffs and strive for selecting the replica that provides the highest overall I/O.

Our default retrieval policy implements a replica ordering algorithm that takes into consideration both the network location and the tier of each replica. The algorithm takes as input (i) the average data transfer rates of the storage media and network devices in the cluster, (ii) the Client location, and (iii) the replica locations and storage tiers. For each replica location and tier, it calculates the rate of transferring the block data from there to the Client. Finally, it sorts the locations based on the decreasing transfer rates.

C. Replication Management

The Master is responsible for ensuring that each block always has the intended number of replicas on each storage tier. The Master can detect the situations of under- or over-replication during the periodic block reports received from the Workers. When a block becomes under-replicated on some particular tier, the Master uses the block placement policy to select one (or more) Workers for hosting the new replica(s). The selected Workers utilize the data retrieval policy for copying from the most efficient location and tier. When a block becomes over-replicated on some particular tier, the Master selects one (or more) replica(s) to remove based on a similar pluggable policy. A user can also explicitly alter the replication vector of a file using the Client. In this case, the same mechanisms are used for achieving the desired change.

D. Multi-level Cache Management

MTFS is a general purpose file system that can be configured to be used as a *multi-level caching system*, offering three unique capabilities. First, unlike typical caching systems, any storage tier above the lowest one can store both base and cached data. Second, the cache management policies can be implemented both inside the system (via pluggable policies) and outside the system (via the Client's API), allowing applications the maximum possible flexibility on how to take advantage of MTFS. Finally, in addition to the typical *cache eviction policies*, MTFS also offers pluggable *cache admittance policies* for deciding when and what data to cache.

E. Interaction with Remote Storage

Another unique capability of MTFS is the ability to attach a remote storage system and get unified view and access methods to all the data. The remote storage can have the form of another distributed file system running in a different cluster (e.g., HDFS, MapR), a cloud-based storage system (e.g., Amazon's S3, Azure Blob Storage), or a network-attached storage solution. When a remote storage is attached, the directory namespace is appended with information from the remote storage in a lazy way, i.e., upon access of data. The user can then interact with the remote storage by using the local namespace through the MTFS Client.

IV. Enabling Use Cases

One of the most powerful features offered by MTFS is the fine-grained control it provides over the various storage media it manages. Applications can explicitly store data in different tiers, change the replication vector of files, and read from the location nearest to them. These capabilities can provide significant benefits to large-scale analytics frameworks (e.g., MapReduce, Hive, Pig, Impala) in terms of manageability and performance as they can schedule their data processing jobs in both a location-aware and a storage-media-aware manner.

MapReduce Task Scheduling: In Hadoop MapReduce [13], the Job Scheduler is responsible for scheduling the Map and Reduce tasks to execute on the compute nodes of the cluster. Since HDFS exposes the locations of the blocks to be processed by the Map tasks, the Job Scheduler will try to schedule the execution of each task to the nearest node containing the corresponding block. With MTFS, the Job Scheduler can also utilize the information about which storage media is hosting each block for improving its scheduling decisions. Furthermore, since the Job Scheduler maintains the task queue, it also knows which files will be accessed soon. The new replication vectors API provided by MTFS allows for the Job Scheduler to implement a *pre-fetching mechanism* and instruct MTFS to start moving (or copying) block replicas to a higher storage tier (e.g., memory). This approach better overlaps I/O with the task processing, which can increase cluster utilization and improve the latency for future tasks.

Workload Scheduling: Analytical workloads are typically expressed as directed acyclic graphs of jobs [13], [1]. In such

workloads, the output data from one job becomes the input to the following job(s), and hence, smart intermediate data placement can have great benefits to the overall workload execution time. MTFS provides the flexibility of placing the intermediate data in local memory or SSDs in order to speed up the overall processing. In addition, the workload processing system has intricate knowledge of any common data among jobs or workloads and can utilize MTFS for dynamically moving data up and down the storage tiers.

Scale-out Analytics for Enterprise Data: Enterprise data such as e-mails, log data, or transaction records often reside in dedicated storage systems with enterprise-level management features. Analysis of such data typically requires expensive ETL processes for copying the data into a separate storage system before running any analytical workloads. The ability to connect remote storage to MTFS has the potential of significantly simplifying the data management by creating a shared-storage back-end system. MTFS will dynamically load data in parallel from the remote storage for processing in the cluster and selectively push data back as needed.

Interactive Analytics: Apart from the typical batch-oriented analytics, *interactive* data exploration is becoming increasingly important. In addition, more complex *iterative* algorithms for machine learning and graph processing are becoming popular [1]. A common aspect for these applications is the need to share data across multiple analysis steps (e.g., multiple queries from the user, or multiple steps of an iterative computation). By allowing explicit memory management, MTFS essentially allows applications to pin their working sets in cluster memory. Fault tolerance is provided by keeping multiple replicas in memory or creating replicas on persistent storage (e.g., SSDs).

V. Related Work

Hierarchical storage management (HSM) systems provide a policy-based way of managing data across a storage hierarchy that consists of disk and tape arrays [14]. The main focus is on archiving inactive files to the lower tiers and retrieving them upon reference. Unlike our system, HSM does not offer any locality or storage-media awareness to applications. *Storage-tier-aware file systems* form the evolution of HSM and are aware of device "types" (arrays of SSDs and HDDs). They offer some control over initial placement and movement of data based on policies [15]. However, each file must reside completely on a tier and they do not offer locality awareness.

Memory caching is a standard technique for improving local data access in distributed applications. *PACMan* [7] is a memory caching system that explores memory locality of data-intensive parallel jobs. However, PACMan does not allow applications to pin data in memory for subsequent accesses. *Resilient Distributed Datasets (RDDs)* are a distributed memory abstraction and a new programming interface for in-memory computation [1]. RDDs allow applications to persist a specified dataset in memory for reuse and use lineage for fault tolerance. They are specialized for iterative algorithms and interactive data mining tools, whereas MTFS offers a general-purpose file system with superset capabilities.

MixApart [3] and *Rhea* [10] focus on improving data retrieval from remote enterprise or cloud storage to local computing clusters. In particular, MixApart utilizes on-disk caching at compute nodes for data residing on enterprise storage systems but cannot write results back to the remote storage. Rhea uses static analysis techniques of application code to generate storage-side filters which remove irrelevant or redundant data transfers from storage nodes to compute nodes. However, the target applications of Rhea are not I/O intensive; they are supposed to have high data selectivity.

VI. Conclusions and Future Work

We presented the design for a novel, distributed, multi-tier file system for cluster computing that is aware of various heterogeneous storage media with different capacities and performance characteristics. MTFS contains automated policies for managing the placement of data across nodes and storage tiers in a cluster. In addition, it exposes the network locations and storage tiers of the data in order to allow higher-level systems to make locality-aware and tier-aware decisions. Our plans moving forward are to investigate the true potential of a tiered storage system across the entire data processing stack.

References

[1] M. Zaharia, M. Chowdhury, T. Das *et al.*, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012, pp. 15–28.

[2] H. Li, A. Ghodsi, M. Zaharia *et al.*, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proc. of the 5th Symp. on Cloud Computing (SoCC)*. ACM, 2014, pp. 1–15.

[3] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *Proc. of the 11th USENIX Conf. on File and Storage Technologies (FAST)*. USENIX Association, 2013, pp. 133–146.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] A. Thusoo, J. S. Sarma, N. Jain *et al.*, "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proc. of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[6] L. George, *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[7] G. Ananthanarayanan, A. Ghodsi, A. Wang *et al.*, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012, pp. 267–280.

[8] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage," in *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, 2011, pp. 25–36.

[9] B. Li, E. Mazur, Y. Diao *et al.*, "A Platform for Scalable One-pass Analytics Using MapReduce," in *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, 2011, pp. 985–996.

[10] C. Gkantsidis, D. Vytiniotis, O. Hodson *et al.*, "Rhea: Automatic Filtering for Unstructured Cloud Storage," in *Proc. of the 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2013, pp. 343–355.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. of the 26th IEEE Symp. on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[13] T. White, *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[14] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *ACM Trans. on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 108–136, 1996.

[15] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami, "Cost Effective Storage using Extent Based Dynamic Tiering," in *Proc. of the 9th USENIX Conf. on File and Storage Technologies (FAST)*, vol. 11. USENIX Association, 2011, pp. 20–34.