A Distributed File System with Storage-Media Awareness

Herodotos Herodotou
Department of Electrical Engineering, Computer Engineering and Informatics
Cyprus University of Technology
Email: herodotos.herodotou@cut.ac.cy

Abstract—**Improvements in memory, storage devices, and network technologies are constantly exploited by distributed systems in order to meet the increasing data storage and I/O demands of modern large-scale data analytics. Some systems use memory and SSDs as a cache for local storage while others combine local with network-attached storage to increase performance. However, 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 storage media (e.g., memory, SSDs, HDDs, NAS) with different capacities and performance characteristics. The storage media are explicitly exposed to users, allowing them to choose the distribution and placement of replicas in the cluster based on their own performance and fault tolerance requirements. Meanwhile, the system offers a variety of pluggable policies for automating data management with the dual goal of increased performance and better cluster utilization. These two features combined 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 devices, and network technologies. Memory capacities are constantly increasing, which is leading to the introduction of new in-memory data processing systems like Spark [1]. On the storage front, flash-based SSDs offer low access latency and energy consumption with larger capacities. However, the high price per GB of SSDs makes HDDs the predominant storage media in datacenters today [2]. This heterogeneity of storage media must be taken into consideration while designing the next generation of distributed storage and processing systems.

Recent work takes advantage of increased memory sizes for improving local data access in distributed applications by using memory caching, storing data directly in memory, or using re-computation through lineage [1], [2], [3]. SSDs have also been used recently as the storage layer for distributed systems, such as key-value stores [4] and MapReduce systems [5]. Finally, [6], [7] focus on improving data retrieval from remote enterprise or cloud storage systems to local clusters by utilizing on-disk caching at compute nodes for persistent data.

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. We present a novel design for a *multitenant, distributed, multi-tier file system (MTFS)* that utilizes storage media (e.g., memory, SSDs, HDDs, remote storage) with different capacity and performance characteristics. Our design focuses on two antagonistic system capabilities: *controllability* and *automatability*. On one hand, the storage media are explicitly exposed to users, allowing them to choose the distribution and placement of replicas in the cluster based on 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 applications to satisfy their requirements. In this way, higher-level processing systems can take advantage of the unique capabilities of MTFS to improve their efficiency and effectiveness in analyzing large-scale data. Finally, in order to support multitenancy, MTFS offers security measures and *quota mechanisms* per storage media to allow for a fair allocation of resources across users.

Our high-level design is inspired by other popular distributed file systems such as GFS [8] and HDFS [9]. We believe this work will open new research directions for improving the functionality of various distributed systems, such as the task scheduling algorithms of MapReduce, the query processing of Pig and Hive, the workload scheduling of Oozie, and 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 network-attached or cloud storage. It is designed to store and retrieve *files*, whose data will be striped across nodes as blocks and replicated for fault tolerance. MTFS employs a *multi-master/slave* architecture shown in Figure 1.

Primary Masters: Each Primary Master manages the *directory namespace*—which offers a traditional hierarchical file organization and operations (e.g., create and delete files and directories)—and maintains the *block locations* that map file blocks to Workers per storage media. Horizontal scalability is achieved by using multiple Masters to form a *federation*.

Backup Masters: Each Primary Master can have a Backup Master that (i) periodically creates and persists a checkpoint of the metadata for increased fault tolerance, and (ii) maintains an in-memory up-to-date image of the directory namespace and is standing by to take over in case the Primary Master fails.

Workers: The Workers are responsible for (i) storing and managing the file blocks on the storage media, (ii) serving read and write requests from the Client, and (iii) performing block creation, deletion, and replication upon instructions from the Masters. The storage media of the same type (e.g., SSDs) across Workers are logically grouped into a virtual *storage*

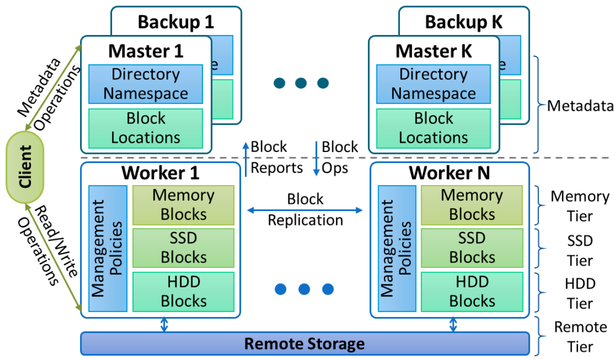

Fig. 1. Multi-tier file system architecture.

tier (e.g., the "SSD" tier). The file blocks can be stored and replicated in one or more tier, based on requests from the Client or pluggable management policies. When remote storage is attached to MTFS, applications can use any of the Workers for reading and writing remote file blocks in parallel. Management policies are used for caching these blocks in one of the higher-level tiers for improving future accesses to these files.

Client: A user or application interacts with MTFS through the Client, which exposes APIs for all typical file system operations. The number of file replicas for each storage tier is specified using a *replication vector* V. For example, 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 in the "HDD" tier. A new API allows users to modify V and achieve various functionalities, including moving or copying a file between tiers, modifying the number of replicas within a tier, and deleting a file from a tier. 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 storage tiers of the replicas, allowing applications to make a fully informed decision for which replica to read from.

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 heterogeneous-aware placement and retrieval policies.

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 obtains a list of <Worker, Tier> pairs for hosting the block replicas from the Master, organizes the pairs in a pipeline, and sends the data. The list is determined using a *pluggable block placement policy*. Our default 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 network-aware data placement is to improve fault tolerance by making replicas across racks, while the goal of tier-aware data placement is to increase I/O performance.

Data Retrieval: When an application reads a file, the Client first contacts the Master for the list of <Worker, Tier> pairs that host the block replicas, and then connects directly to the Workers for reading the data. The list is ordered based on a *pluggable data retrieval policy*. Our default policy takes as input (i) the average data transfer rates of the storage media and network devices in the cluster, (ii) the Client's network location, and (iii) the replica locations and storage tiers. For each replica, it calculates the transfer rate to the Client and sorts the list based on the decreasing calculated transfer rates.

Replication Management: The Master is responsible for ensuring that each block always has the intended number of replicas on each storage tier. When the Master detects situations of under- or over-replication during the periodic block reports received from the Workers, it utilizes the placement and retrieval policies for creating or deleting replicas.

IV. Enabling Use Cases

The fine-grained storage control MTFS provides offers 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 addition to location, a MapReduce Scheduler can exploit the tiering information of each block for making better scheduling decisions. Furthermore, the Scheduler can implement a *pre-fetching algorithm* and instruct MTFS to start moving (or copying) block replicas to a higher storage tier before scheduling the tasks.

Workload Scheduling: Analytical workloads are typically expressed as directed acyclic graphs of jobs [1], [3]. Intermediate and common data between jobs can be dynamically placed in higher storage tiers to speed up the overall processing.

Scale-out Analytics for Enterprise Data: 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 [7]. With MTFS, all storage tiers can be used for caching enterprise data as well as hosting intermediate data.

Interactive Analytics: By allowing explicit memory management, MTFS allows interactive applications to pin their working sets in cluster memory. Fault tolerance is provided by keeping multiple replicas in memory or on a lower tier.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma *et al.*, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. of NSDI*. USENIX, 2012, pp. 15–28.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proc. of SOCC*. ACM, 2014, pp. 1–15.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proc. of NSDI*. USENIX, 2012, pp. 267–280.

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

[5] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-pass Analytics Using MapReduce," in *Proc. of SIGMOD*. ACM, 2011, pp. 985–996.

[6] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. I. Rowstron, "Rhea: Automatic Filtering for Unstructured Cloud Storage," in *Proc. of NSDI*. USENIX, 2013, pp. 343–355.

[7] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *Proc. of FAST*. USENIX, 2013, pp. 133–146.

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

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. of MSST*. IEEE, 2010, pp. 1–10.