A Survey of Distributed Object Storage Solutions

by

Saksham Goel, Nilanjan Daw, Rinku Shah

under the guidance of

Prof. Umesh Bellur & Prof. Purushottam Kulkarni

Department of Computer Science and Technology
Indian Institute of Technology, Bombay
Mumbai 400 076

This work is supported by a research grant from Huawei Technologies vide grant number RD/0120-HUAWE18-002.

Contents

List of Figures

Chapter 1

Introduction

High performance computing (HPC) is the practice of aggregating computing resources across systems in a way so as to deliver a consolidated performance higher than that achievable from discrete workstations or servers. These HPC systems find usage across a large spectrum of fields like that of engineering, science, business. It finds usage in running distributed algorithms for climate research and weather prediction [26, 28, 27, 9], oceanic studies [7, 1], machine learning [20, 19, 24], Big Data processing, bioinformatics, drug research and modeling. For instance, during the ongoing Corona-virus pandemic, HPC clusters and distributed systems in general has been at the forefront for searching of novel drugs for the disease and genetic material sequencing. A typical setup for such a system involves a distributed application or algorithm which uses a computes and storage nodes distributed across a cluster to work on high dimensional problems.

The need for a storage system in the HPC ecosystem for storing of large datasets usually associated with such algorithms, necessitates the presence of a large storage with extremely low read and write latency. Today's distributed setups have a memory storage hierarchy that offers different characterizations in terms of performance, capacity and functionality. Recent advances in storage and network technologies like Persistent Memory or Non-Volatile Main Memory (NVMM), Remote Direct Memory Read (RDMA), SmartNICs has made it possible to bring these two orthogonal issues together. For example, NVMMs provide extremely low latencies, close to DRAM read latency, while RDMA capable NICs allow direct access to remote system Main memories bypassing the host OS completely.

In this project, we explore design options of distributed file systems, and as a more generic functionality of an object store, that can push the limits of latency and throughput exploiting the heterogeneous memory-storage hierarchy.

Chapter 2

Survey of Techniques

The main components of a storage layer are (i) a metadata management unit which defines how the metadata related to the data stores are managed, (ii) an optional cache management layer for faster lookup, (iii) memory and Addressing Management to support multiple storage technologies, (iv) a Transaction Management subsystem for consistency guarantees, (v) a Consensus mechanism among the distributed servers and (vi) distributed locking mechanisms for concurrent accesses. Figure 2.1 shows a high level view of the components discussed here and their relative position in the software stack.

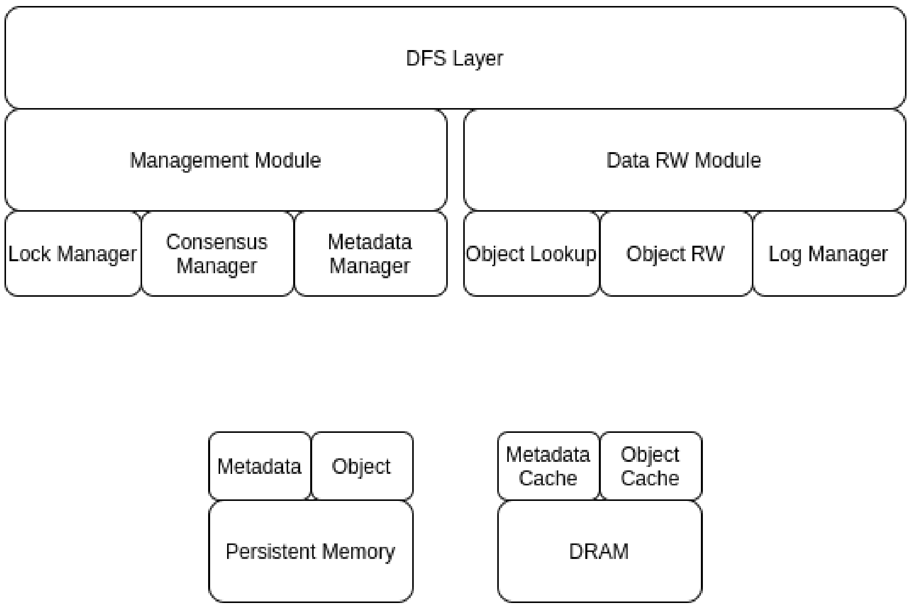


Figure 2.1: High level component Diagram for a Distributed File System

Figure 2.2 illustrates a web of how different solutions address these issues. In the next subsections, we dig deeper into each storage component and explore how challenges in those components are addressed in the present state of the art.

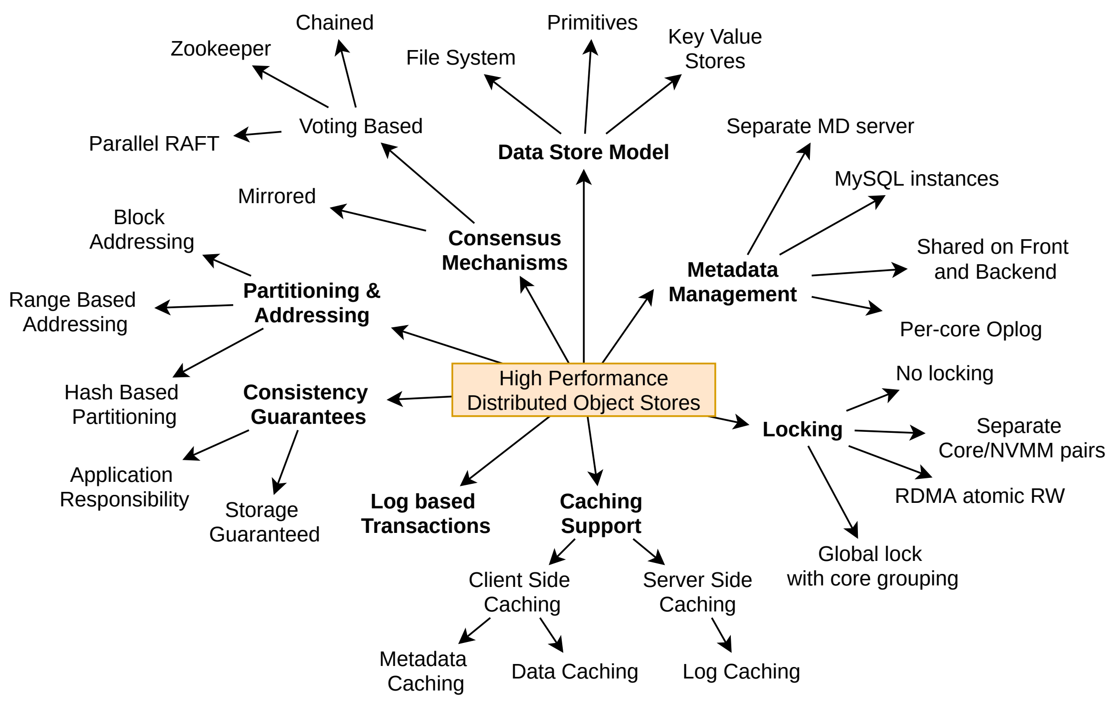


Figure 2.2: Classification of Non-volatile memory and SmartNIC based storage solutions.

2.1 Data Store Model

The Data Store Model describes the way in which data is represented complete with its relationships and the way in which it can be accessed. Accordingly, we look at two popular paradigms: Key-Value stores in which data is represented as independent pairs of keys and values, and File Systems which focus on contiguous runs of data blobs, generally larger than those associated with KV stores. We also consider data-model independent systems which provide a set of primitives to implement any model on top. Figure 2.3 shows the distribution of storage solutions in this space.

2.1.1 Key-Value Stores

The need for key-value stores arises predominantly from web applications which need to store a large number of relatively small sized records [23]. Because of this, most Key-Value stores support values which are $<$1MB in size, treating them as a single indivisible chunk of data identified by a unique variable-length key [8, 5, 21, 18, 6]. Several auxiliary structures such as version numbers [18] and counters [6] may also be associated with each pair depending on the consistency model used.

The value is most often treated as an uninterpreted blob of bytes with no assumed internal structure [6, 18, 8, 21, 4]. Moreover, the pairs are treated either completely independently with no relational-model [6, 8, 21, 4] or segregated into structures like

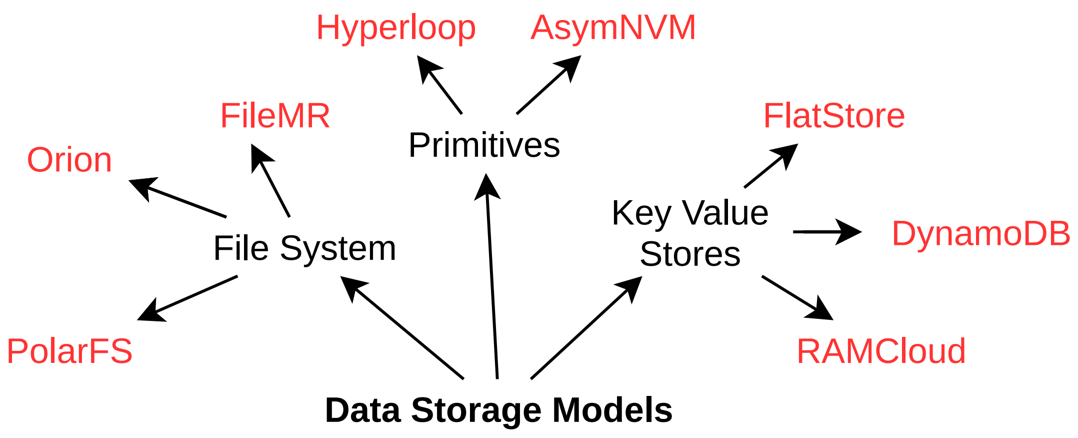


Figure 2.3: Web of Data store models

tables [18] or containers [13]. Accordingly, this class of object stores provide a restricted query set mainly consisting of read, write and update of a single pair at a time. Some systems extend this set to support querying over a range of data, leveraging batching and caching to improve performance [18]. Another class of key-value stores known as document stores require the data to follow a specific representation and allow for simple queries based on stored values [5], often at the cost of speed or relaxed consistency guarantees.

2.1.2 File Systems

The File system data store model is concerned with storing large contiguous blocks of data which are generally accessed and modified sequentially, as opposed to random accesses common in KV stores. The data blocks are typically stored under a file like abstraction and support hierarchical structuring. Most File Systems only support sequential read/writes, with writes being persisted on a secondary storage device or (persistent) memory. The sequential access property of these data stores opens up avenues for optimisation which is generally not available to random access KV stores. In the present text we look into Orion [30], PolarFS [3] and FileMR [29] as background in this space.

2.1.3 Primitives

Thus far the fully-fledged data storage systems discussed are either optimised to cater to small key-value pairs or sequential data-blocks. However, certain application needs might not be fulfilled by these systems. In such cases it is imperative to provide low-level storage and synchronisation primitives which can be leveraged by higher level applications to build their own storage systems, catering to their consumer's niche requirements. Hyperloop [11] provides a set of such synchronisation primitives achieved by leveraging RDMA-based read/writes to facilitate distributed transactions by userspace applications. AsymNVM

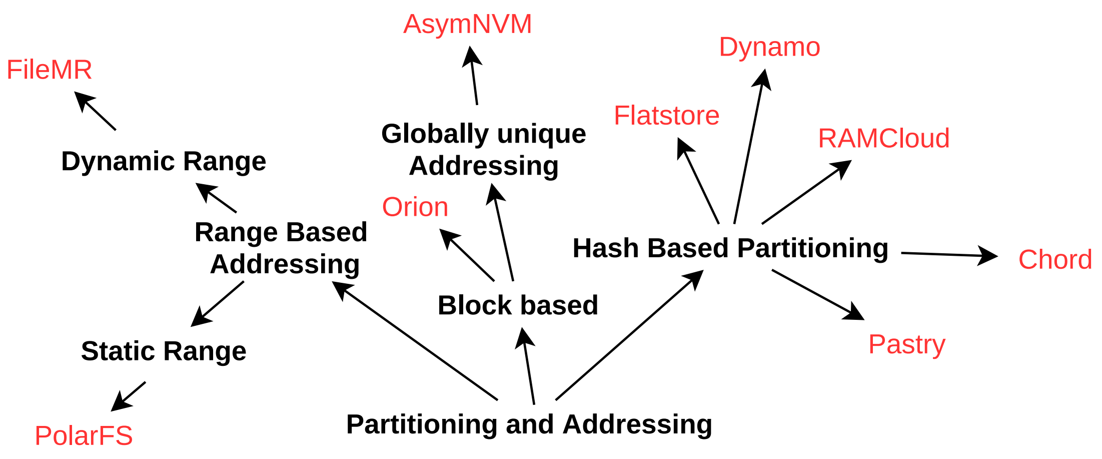


Figure 2.4: Solution Web for Data Partitioning Schemes

[15] iPipe [14] discusses a similar set of primitives and outlines how they can be leverage by various distributed data structures.

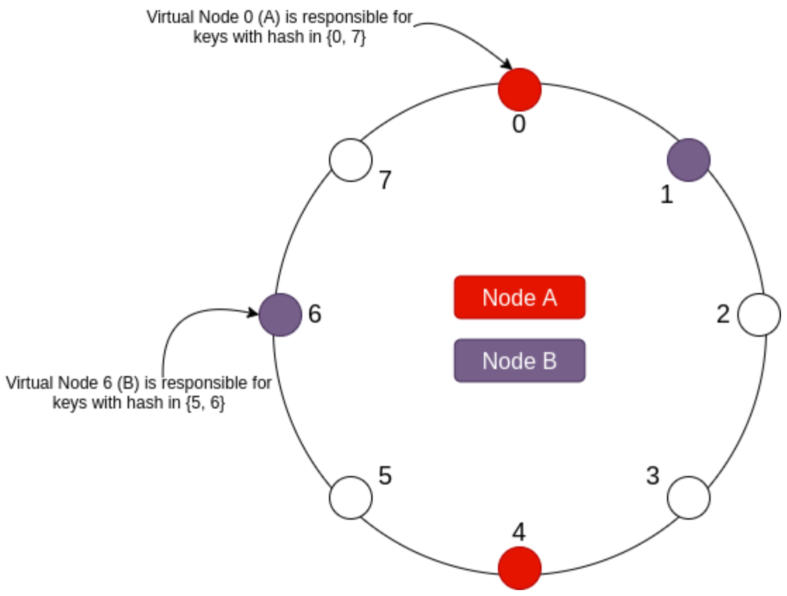
2.2 Data Partitioning

Data Partitioning refers to techniques to divide data across multiple systems, typically to enhance scalability by distributing the service load. In Key-Value stores, key-oriented partitioning techniques are dominant given that each key can be accessed independently. On the other hand, File Systems usually resort to range-based partitioning schemes presumably to leverage spatial locality that comes with file access patterns as shown in Figure 2.4.

We describe both these sets of techniques and several of their variants below.

2.2.1 Hashing

Data can be distributed among nodes based on a hash generated from an identifier - generally the key in KV stores, or the file's path in a file system. These algorithms are usually static, and hence enable fast lookup by any participating node locally. This also obviates the need of a central manager and reduces network load. An important drawback is that these algorithms usually do not account for dynamic changes in the query workload. Hashing based partitioning algorithms also suffer in performance when there is movement of data, for example to recover from node failures, as this requires re-calculation of the hash for a significant portion of objects. A trade-off between lookup speed and hashing overhead can be seen based on the granularity of data size hashed, with some systems choosing to hash each key-value pair independently [8, 6, 13] and others hashing larger chunks of related data together [18].



Virtual Node 0 (A) is responsible for keys with hash in {0, 7}
0
1
2
3
4
5
6
7
Node A
Node B
Virtual Node 6 (B) is responsible for keys with hash in {5, 6}

Figure 2.5: Consistent Hashing: Hash Ring with Virtual Nodes

Consistent Hashing considers the range of the hashing function as a ring such that each storage node gets assigned an arc of this ring. This arc distribution is contiguous so that each possible hashing key is mapped to exactly one node. A key benefit of consistent hashing is that the hashes don't change with addition or removal of storage nodes. Further, this ensures a relatively equal distribution of pairs amongst the nodes [10] assuming a non-adverserial workload and universal hash functions.

To deal with heterogeneity in processing power and capacity of different nodes, *virtual nodes* are used to assign non-contiguous arcs on the hash ring. Each physical node then maps to multiple virtual nodes [6, 12]. While virtual nodes provide better distribution, they result in a more complex metadata management and higher movement during failure and addition of nodes; most such systems therefore limit the no. of virtual nodes allowed. A host of other randomization techniques with refinements based on geo-location and network latency are also used for better balancing [18].

Consistent Hashing is also used along with *Distributed Hash Tables*, in which data mapping is stored in a decentralized manner [25, 22]. These generally require more than one hop to locate data, owing to the fact that each node only contains a subset of the routing information.

In a hyperconverged setting, where each compute node doubles as a storage node, the responsibility of calculating the hash during the initial write is sometimes taken up by the client itself [13]. This also opens up the opportunity to encode hints within the generated key regarding the preferred storage location for the value and other optimisations.

2.2.2 Range-based Partitioning

A common trend observed among most storage systems has been the presence of a central metadata management server (MDS) for address management. This puts the MDS on the critical path for each read and write. While the read cost can be partially offset using efficient metadata caching, writes still must go through the MDS. File systems, due to their contiguous memory allocation, provide a unique opportunity to reduce the write cost by optimizing block sizes. In most file based storage systems, the MDS allocates memory chunks to the backend servers which are used to serve multiple writes. PolarFS[3] uses chunk sizes of 10GB to minimise metadata management. These chunks are further subdivided into smaller blocks of 64KB inside the ChunkServers. The ChunkServers store the logical block address to physical address mapping together with a bitmap of free blocks.

FileMR[29] provides a Range Based Address translation mechanism using the Memory mapping table of RDMA DAX. This allows FileMR to address large contiguous memory spaces in the NVM address space and leverage extent based file system mechanisms. This also decreases the memory required to store the address mapping and the time required to perform lookups on the table.

Flatstore[4] on the other hand uses a Hoard-like memory allocator for logs. It divides the NVM storage space into 4MB chunks. Based on the class of memory required this 4MB chunk can be divided into subblocks of different sizes. The type of block used, the size of each block, and the list of free pages are stored at the start of each 4MB chunk. The chunks are also allocated to dedicated server cores to avoid locking induced latencies.

2.3 Consistency

Nodes in a distributed system are likely to fail due to a host of issues such as power failure, damage to hardware, and even natural disasters. This makes it important for systems to replicate their data across multiple separated nodes for better durability. Replication across nodes require careful use of consensus protocols to maintain a single global view of data. Consistency models account for the effect of concurrent requests and consequently the state of a distributed object store as viewed by clients in the presence of multiple replicas hosting the same dataset. Figure 2.6 shows an infographic of various techniques used to ensure consistency and node consensus.

We describe different consistency models and techniques following this categorization.

2.3.1 Strong Consistency

Orion[30] relies on a central metadata server which acts as the single source of truth for all metadata related information. This makes it critical to the operation of the storage system and any corruption in the data can lead to its failure. To prevent this, the MDS

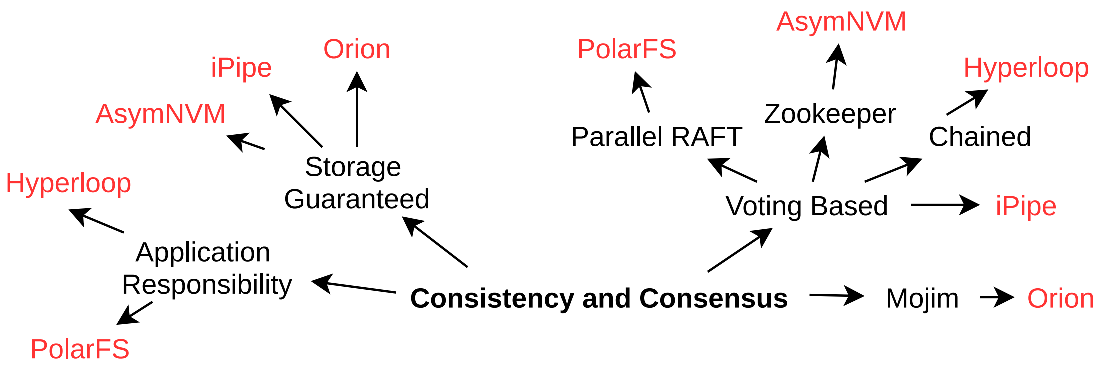


Figure 2.6: Solution Web for Consistency and Consensus Algorithms

can be made to operate in a high-availability mirrored cluster using Mojim protocol, where the master receives all write requests which are mirrored on the replica before an ACK is sent back, which takes over in case of a master failure. On the other hand if a Data store fails and rejoins after sometime, it rebuilds its data store by reading the pages committed during its absence from its peers in the replicaset.

AsymNVM[15] on the other hand uses Zookeeper as its consensus mechanism, which uses a lease based system to detect liveness of nodes in a replicaset. A lease which is not renewed before its expiry indicates a node failure. In case of front end failures, the backend informs the frontend of any log inconsistencies which then re-executes the Operation log stored on its system. However in case of a backend failure, after rebooting the backend will reconstruct the virtual to physical system address mapping from its NVM storage and inform the frontends of any inconsistent transactions which will then be fixed using redo operations from the frontend.

The DAOS system[13] supports both strong and weak consistency protocols. If the DAOS system is run with strong consistency guarantees then the storage server nodes run RAFT protocol to replicate data amongst themselves. Even though this enables strong consistency guarantees, it comes at the cost of increased latency and throughput overheads.

M. Liu et al[14] describes a a multi-Paxos based algorithm to run a replicate-set on a smartNIC. While the consensus protocols discussed by Orion and AsymNVM require the intervention of a conventional server host, iPipe's multi-Paxos implementation bypasses such a requirement, running exclusively on the NIC. This reduces computational load on the server system while at the same time receives better packet latency owing to the NIC being closer to the network.

2.3.2 Weak Consistency

Traditional stores like Dynamo [6] provide eventual consistency to achieve high availability. This model guarantees that all the replicas will *eventually* converge to a consistent state.

Further, in such systems, not all replicas need to be consistent. A sloppy-quoram in which each read and write consults R and W nodes (R+W $>$ N, where N is the replication factor) respectively can ensure a consistent view of the data stored. This also means that reconciliation of different versions of the same key must be done at reads.

Raft is a common strong consistency protocol used by various distributed systems. However, Raft is suitable for serialized write and hence can lower system throughput. ParallelRaft [3] relaxes the log management such that when a log commit reaches a follower node, all previous logs may not be committed. This is tolerated by delaying ACKs from the followers until after all the previous logs in its memory has been successfully committed. The notion of serial commit of logs in the Raft leader is also relaxed such that any log entry can be committed, irrespective of its previous log's status, as soon as ACKs from a majority of previous nodes has been received. PolarFS [3] also employs a new data structure named the *look-behind buffer* to bridge any possible holes in the commit log created due to out-of-order log commits. In case of a node failure, if the difference between the leader and the follower is only a few entries, a fast-catch-up mechanism is used where the holes between the leader and follower nodes are filled using the look-back-buffer by directly copying them from the leader. However in case of catastrophic failures, a full log rebuild is required. In a similar vein, while dynamo [6] does provide merging of multiple versions of the same key in which the versions are totally ordered, it leaves partially ordered conflicting version reconciliation to the application

Hyperloop[11] uses a similar chain of reasoning regarding consistency management. Since their solution is to provide primitives for higher level application development, the authors describe a chained replication system for data replication however leaving consistency maintenance to the higher level applications.

The DAOS system[13] as already mentioned can run in either weak or strong consistency mode. In cases where the client side application can withstand write losses or prefers to replicate data themselves, the DAOS server side suspends proactive replication, with the client becoming responsible for data replication on the target nodes. A side-effect of client side replication is that even though it can detect node failures and prevent data loss in case the failure happens before a write, it is possible for write losses to happen in case of failures occurring during write replication.

2.4 Metadata Management

A general trend observed in most File system and a few KV stores is the presence of a central metadata management unit which stores metadata regarding a data blocks like their location in the distributed system, ownership and access control rules, access patterns etc and acts as the single point of truth for the storage system. Clients requesting for data blobs general perform an initial lookup on the metadata server regarding the location of

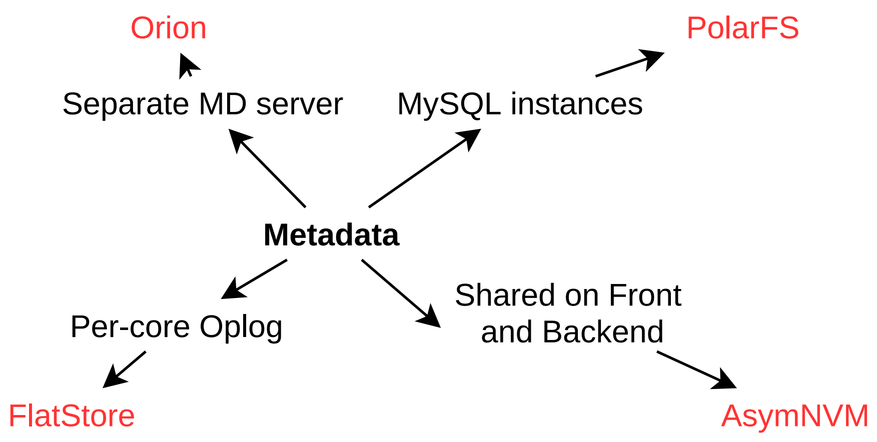


Figure 2.7: Solution Web for metadata management techniques

the block before sending the actual IO request, thus putting the metadata server on the critical path on a significant slice of IO requests. Figure 2.7 shows an overview of various metadata management techniques.

Orion [30] clusters consists of a *Metadata Servers* (MDS) and data storage units (DS). The metadata servers run on a high-availability cluster with the servers storing the authoritative copy of the operation logs. The servers also store a bitmap of free space across the cluster. Data stores register memory needs with the MDS, which are allocated in large chucks to avoid frequent look-ups. [3] follows a similar architecture, terming their metadata management unit as the *PolarCtrl*. The PolarCtrl is tasked with maintaining cluster membership information of Data servers called *ChunkServers*, maintaining volume information, creating and allocating memory chunks, synchronising metadata and performing sanity checks to prevent data corruption. PolarCtrl uses a MySQL instance as a metadata repository. On the other hand, [15] follows a shared model for metadata management. The backend nodes store the metadata in hashtable based data-structure for for fast lookup. The metadata comprises of log addresses, NVM persistent data, root addresses of datastructures and datastructure locks held during transactions. The frontend nodes cache the address translation hashmap in local memory, with a hybrid LRU and RR based cache eviction policy. Chen et al [4] follows a similar strategy for small KV updates, which are directly written to the OpLog along with the indexing metadata, while a persistent storage allocator handles space allocation and storage of large Key values. The logs are kept separated on a per-core basis to avoid locking and contentions among cores. [13] supports a POSIX File System like abstraction using a shim on top of their object store API. They mitigate the need of running a central metadata server by directly encoding metadata information inside the 128bit key generated to identify each object stored in the DAOS system.

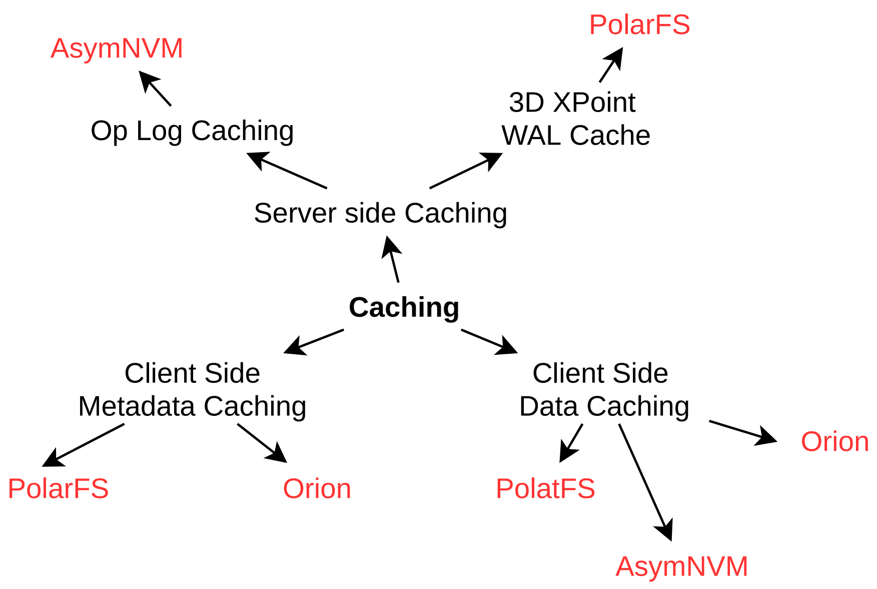


Figure 2.8: Solution Web for various Caching Techniques

A significant source of latency in a centralised metadata stores such as [30, 3] is the key lookup cycle associated with every query. With rise in IO pressure from client nodes, the central metadata server can become the source of a bottleneck, since every IO needs to pass through it.

2.5 Cache Management

Network remote read writes are comparatively slower than local IO, and hence most file systems prefer a local memory read/write. An efficient caching policy on the client system can help improve local IO hit-rate and in turn the storage latency and throughput. Furthermore as discussed earlier, the metadata management layer is on the critical path for most of the IOs, and can become a potential source of IO bottleneck. However this can also be partially mitigated with effective caching at the client nodes. Figure 2.8 illustrates various client and server side caching techniques across solutions.

Orion[30] performs both data and metadata caching at the client node. For every read Orion caches the metadata and the data to its local DRAM, with an LRU based eviction policy. Clients use these local caches and migration with copy-on-write to prevent unnecessary remote reads.

PolarFS[3] operates in a similar manner, metadata information is cached on the *PolarSwitch* to avoid lookups on the IO critical path. The mapping tables of chunks occupy around 640KB memory which can be easily accommodated in the main memory of the chunk server. During a lookup, a *PolarSwitch* finds out the location of *ChunkServer* replicaset using its local metadata cache. The metadata information is synchronised by the *PolarCtrl* between itself and all the *PolarSwitches*. This model also prevents disruption during small outages in *PolarCtrl*'s availability. The *ChunkServer* on the other hand uses a small 3D XPoint SSD buffer as a write cache for WAL, this improves write throughput for the servers.

AsymNVM[15] is built on the principle of efficient batching and caching to provide large throughput and low latency. To increase the throughput of the system, AsymNVM uses an operation log (OpLog) to cache data from write operations. These are then batched together and applied to the data structure enabling larger throughput for RDMA writes. For reads, data blocks are first checked for their presence in the local cache (DRAM based). In case of a cache miss, for cold data, the data is directly read from the Backend store using one-sided RDMA_read, however for frequently accessed data like root for a tree, the data is swapped into the cache from the remote backend via RDMA_read and then read from the cache. AysmNVM uses a LRU and Random Replacement(RR) based hybrid cache eviction policy, whereas a random set of pages are first selected and the least recently used pages from the set are evicted from the cache.

DAOS[13] arranges its storage nodes in a tree-like hierarchy with sibling nodes representing shared fault levels, for example DAOS agents sharing the same motherboard will be at the same fault level. DAOS uses incast variables to invalidate server side caches in such a tree-like hierarchy. Each node on a cache write sends a cache invalidation trigger to its parent which in turn invalidates the cache of its parent. Once the root is notified on such a cache invalidation, it sends triggers to all other branches in the DAOS system. For key-values with a size less than a pre-defined threshold, DAOS sends the data point along with the cache invalidation RPC, while for larger KVs, target nodes directly reads the data point from the source node using RDMA.

2.6 Transaction Management

Data integrity management, consistency and crash recovery are important parts of a datastore. Most filesystems discussed till now provides some form of transaction management system. The commmon method for ensuring transactions across most of these solutions is using log based transactions and write-ahead-logs (WAL).

Orion[30] has a configurable consistency policy, where as it can weaken the consistency guarantee by forwarding a speculative log write to the Metadata Server before a write actually happens on the filesystem. For strong consistency however, Orion delays the log update until after the write happens on the filesystem. To perform this over RDMA network calls, Orion sends the global address along with the packet. This is then used by the DS to send back an acknowledgement to the Metadata Server which then processes the log entry.

Hyperloop[11] provides a group-based transaction system which works entirely via the NIC. To provide durable writes it uses an ACK sent from the last node of a replicaset to act as an indicator for successful write. Since an ACK is sent as soon as the data reaches the NIC memory and not the actual host storage or memory, a 0-byte READ is sent immediately after the WRITE request is made to force the data to be flushed from the NIC memory. This allows the ACK to actually confirm a successful write. Hyperloop uses this technique to provide group based write(gWRITE), flush (gFLUSH), compare and swap(gCAS) and other primitives.

AsymNVM[15] uses two separate logs to ensure data consistency. The lower level memory logs are generated with each write operation on the backend nodes which ensure they are written in an all or nothing fashion. Only after the memory log has been persisted is an ACK sent back to the frontend for the write to complete. However since singular writes over a network is costly, IO operations are batched together on the frontend nodes. This is done using the Operation Log. Once the log has been persisted it can be used to replay all operations done on the datastructure and hence the actual data write to the backend nodes can be delayed.

FlatStore[4] also similar operation log based batching techniques to enable higher network throughputs. However, they observed that storing large data in the log reduces opportunities for batching. Hence, extremely small KVs ($\tilde{2}$56B) are stored directly on the log where as larger values are stored via pointer references. To further improve batching of logs, Flatstore performs horizontal batching of Oplogs, where each core tries to acquire a global lock, which it then uses to batch logs across cores for final transfer to the backend.

DAOS[13] tags each IO operation with a 64-bit epoch based timestamp, which is a combination a logical and physical clock. The epochs are used to serialise concurrent writes. The DAOS transaction API allows multiple object updates to be coalesced into a single transaction, with each update tagged with the same epoch. However, DAOS provides no ordering guarantees among writes within the same transaction, with write replication happening in any arbitrary order among target replicas. In case of concurrent writes with the same epoch timestamp, DAOS provides no consistency guarantees, with the client only being warned of such overlapped writes.

2.7 Concurrent Access and Lock Management

Consistency guarantees also require parallel or concurrent access be properly managed in order for distributed systems to work without data corruption. This requires the need for critical sections in a data block to be locked before a write can occur. Broadly two parallel trends has been observed in this space, the first being that of lock based access control where critical section data is locked via a distributed shared lock, and the second being that of lockless accesses via contention avoidance techniques and multiversion datastructures.

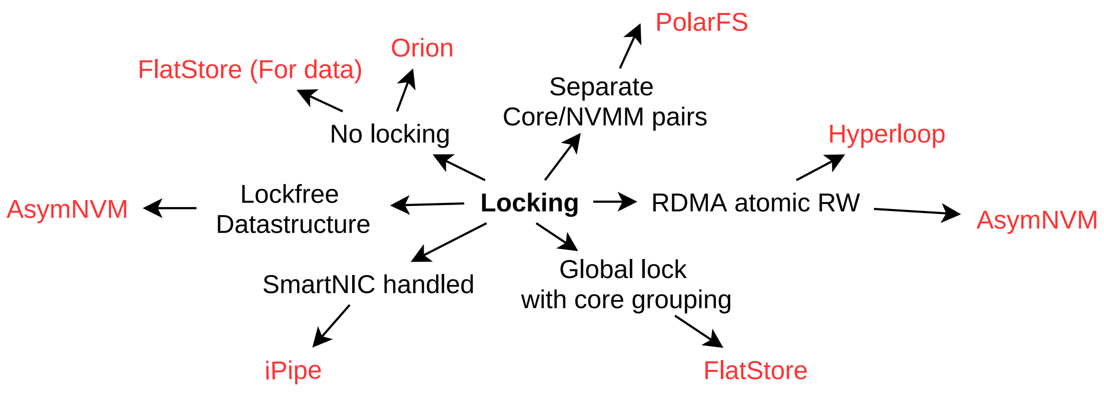


Figure 2.9: Solution Web for various concurrency control mechanisms

Figure 2.9 shows various concurrency control mechanisms in today's state of the art.

2.7.1 Lock Based Concurrent Access

Hyperloop[11] makes use of RDMA atomic Read writes to provide locking based protocols for higher level systems to use. They use the techniques already discussed in transaction management to provide systems with group based locking mechanisms without involving the CPU.

AsymNVM[15] discusses both lock based and lock-free data structures. It uses RDMA compare and swap atomic operations as a writer lock. The lock is defined alongside the root of the data structures. The lock is released only after all writes have been completed. Each writes atomically increase a Sequence number(SN) twice, once at lock and once at unlock. Reads can only occur if it can acquire a lock when the SN is odd else a backoff policy is used. Reads also need to check if the SN is same as it was before it started to the read to fend off data inconsistencies.

FlatStore[4] uses a global lock to prevent contention among cores when trying to acquire Oplogs for horizontal batching. To reduce overheads, the leader with the lock releases it as soon as it acquires the operation logs. To further reduce locking overheads, the authors propose grouping cores belonging to the same socket for horizontal batching considerations and by extension locking.

iPipe[14] on the other hand performs transactions bypassing the host CPU. A coordinator reads the data into its memory, validates and sets a version lock. It updates its local lock and sends the data to the other participants for replication. Once the participants reply, the lock is released and the write is deemed successful.

2.7.2 Lockless Concurrent Access

Orion[30] uses Client Side arbitration to avoid locking of data. It is built on the principle that RDMA inbound reads are lighter than outbound writes, and CPU cycles are costly for the metadata server. The MDS thus appends log commits atomically whenever a request comes. The clients on the other hand detects any mismatch between its local copy and the MDS log copy before issuing write requests, and updates logs accordingly. In case of mismatch happening due to concurrent accesses, the client fetches all log entries following a sync point, and rebuilds the log entries in its DRAM and re-executes the user requests.

The FlatStore [4] architecture mandates a private storage for each server CPU core. Thus IO requests to CPU cores are sequentially transferred to the assigned storage node. Thus data blocks avoid contentions from parallel accesses. This obviates the need for data block locking during read writes.

AsymNVM [15] discusses a lock free mechanism for concurrent access using multiversion datastructures. The writer copies all affected nodes to create a new copy, updates their pointers and inserts new data into the copied node. Finally, when all datastructure writes are satisfied, the writer atomically changes the root of the tree to point to the new datastructure. Since the pointer change is atomic, no crash recovery or locking is required, since a failure will only result in the latest write being lost without leaving the system unstable.

DAOS [13] provides optimistic concurrency control using multi-version timestamp based concurrency ordering. This lets the DAOS system avoid any locking based concurrency bottlenecks at the cost of providing no worse case concurrency guarantees. In case of writes having the same epoch timestamp, the DAOS system detects such violations and informs the client so that they can be retried.

2.8 Rounding up

We discussed a host of solutions in the distributed storage system space with regards to different storage system components. Figure 2.12 provides a summary of the solutions we discussed in the present report and their relative positions in the DFS problem space.

Over the solutions we discussed, some trends can be observed. RDMA has been found to be used overwhelmingly by almost all solutions for purposes ranging from network communications for consistency protocols, to replication and metadata and data transport, along with a cluster of other communication protocols like SPDK, PMDK, Infiniband as can be observed the solution web Diagram 2.10. RPC based communication protocols have also been observed to be used considerably across multiple architectures mainly for metadata communication and transport. On the other hand, we note a general push for storage systems to employ byte-addressable non-volatile storages like NVMM and

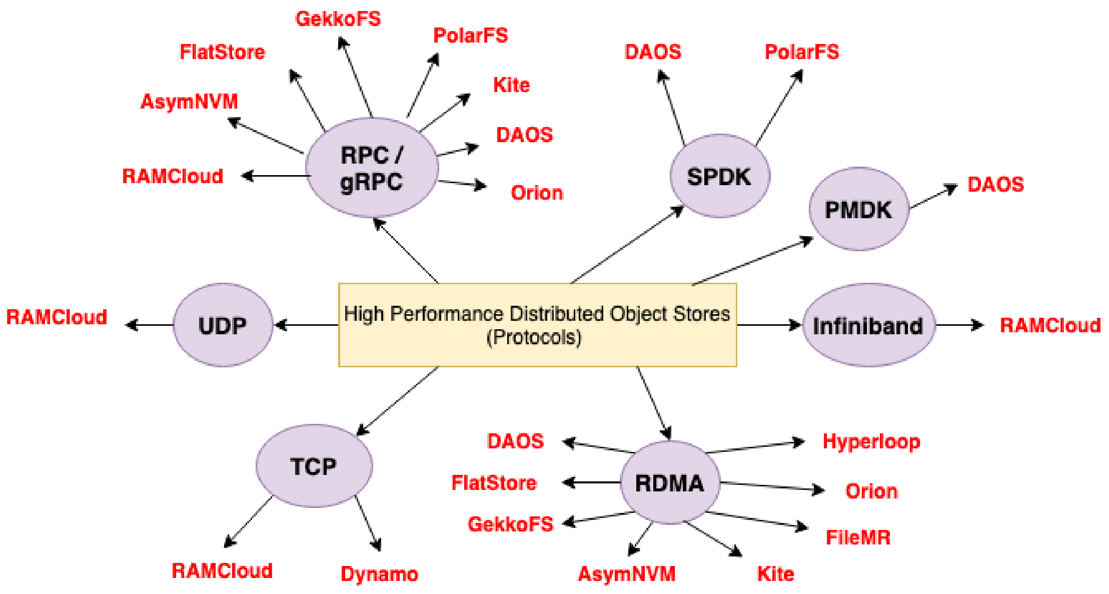


Figure 2.10: Solution web diagram of communication protocols used by distributed storage systems

NVMe SSDs over traditional spinning disks. Recent advancement in non-volatile storages have reduced prices considerably making such a push possible. DRAMs are being used as caching layers in most LFS based file systems for faster data access. We also observed some recent development which exploits NICs for accelerating storage systems. The solution web Diagram 2.11 captures these trends observed across solutions.

We also observed a general trend for Key-value based storages to use hashing based addressing schemes for generating key-value storage locations, while file based storages tend to use range based or block based addressing with a centralised metadata server acting as an arbiter. A hash based addressing scheme provides faster random KV lookup compared to a centralised addressing schemes but runs the risk of storage bottlenecks, while a block based addressing scheme provides better storage loadbalancing at the risk of the metadata server being a potential access bottleneck. The risk of the storage server and the metadata server being a bottleneck is partially offset by client side data and metadata caching.

On the topic of replication of data for fault tolerance, consensus amongst replicasets and transaction and locking support for concurrent data access, we observed a large spectrum of technologies being used. Most of the consensus mechanisms are based on voting based techniques, with some consensus protocols like ParallelRaft relaxing strict consistency requirements by offloading some responsibilities userspace applications. Transaction management has been overwhelmingly log based with almost all solutions with transaction support using a log structure mechanism. While some solutions tend to avoid data struc-

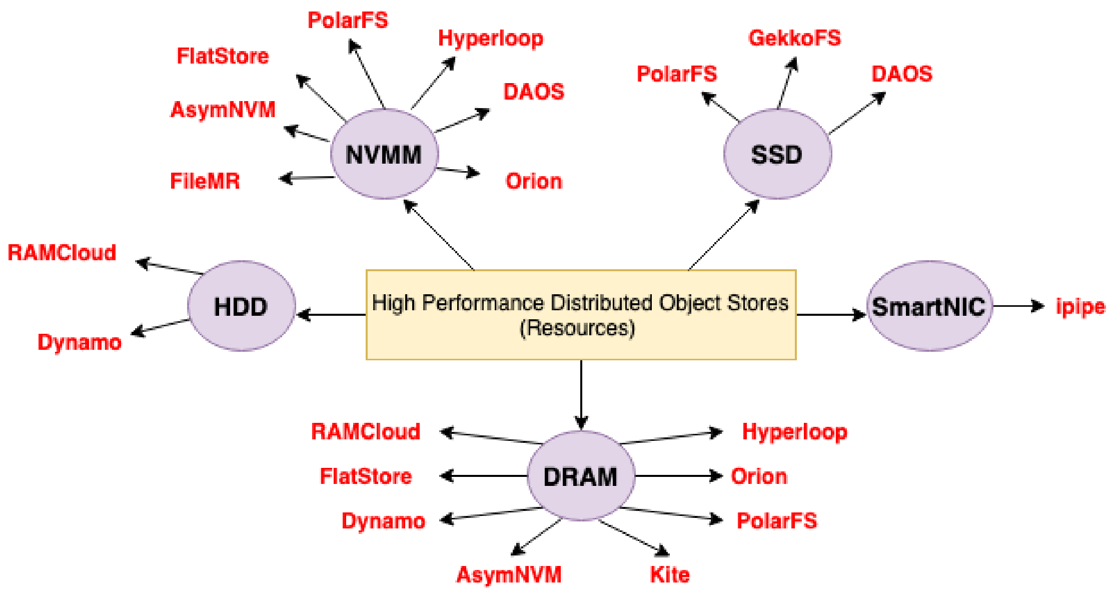


Figure 2.11: Solution web diagram of hardware resources employed by distributed storage systems

ture locking using contention free techniques like per-core request queues, multi-version data structures, others have shown to provide distributed locking mechanisms exploiting the atomic read/write guarantees of RDMA.

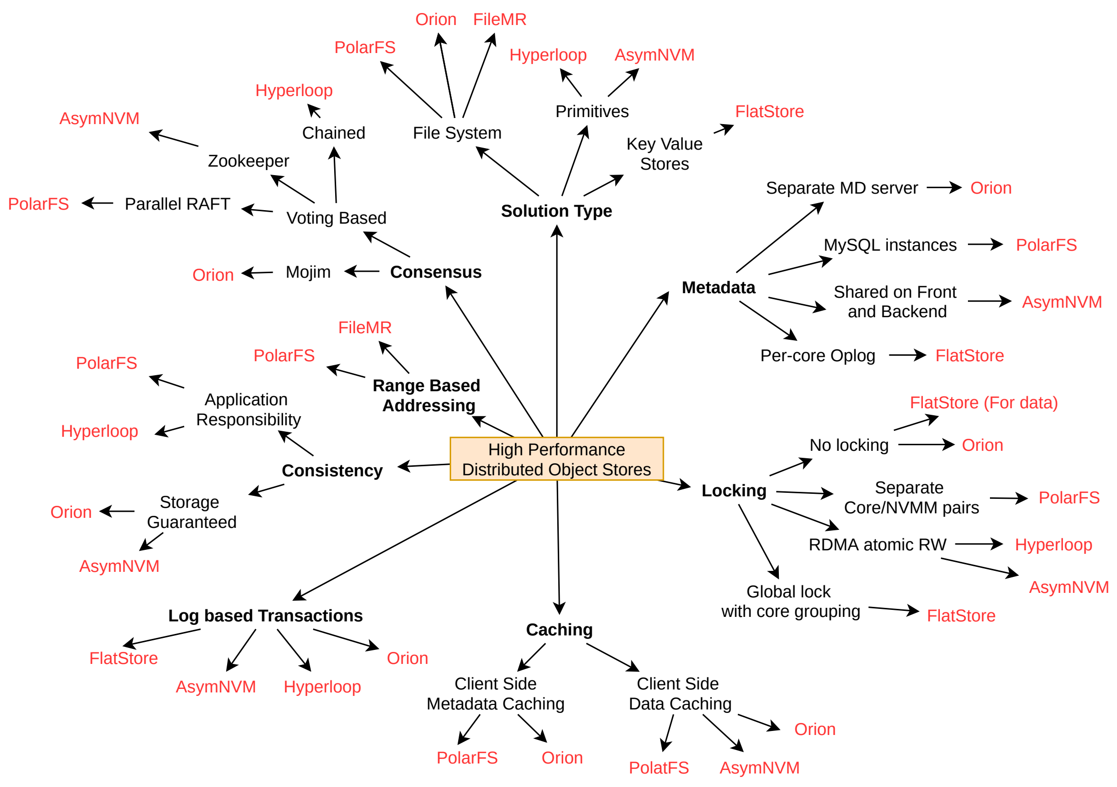


Figure 2.12: Solution web diagram for DFS components

Chapter 3

Directions for Exploration

The advent of smartNICs have opened up new exploratory avenues in distributed storage system design. They provide specialized compute and storage at a unique position in the network stack. Given that each component of a distributed store, – prefetching, caching, provisioning, consistency, fault tolerance etc. – requires logic processing and state to run, any combination of them can be offloaded in part or completely to these next-gen NICs. This allows for a myriad of possibilities, each of which affect performance. Navigating across all options, a careful consideration of each possibility is required to optimize for specific criteria and requirements.

In the following sections, we first describe these technologies in detail with the aim to understand exactly what they offer. Then, we discuss what we believe are interesting approaches to leverage these technologies by modifying existing designs.

3.1 Benchmarking

SmartNICs are generally equipped with a significant number of general-purpose compute cores (typically more extensive than those present on a host) on the NIC along with a smaller scratchpad memory. This brings the storage operations closer to the network while at the same time reducing resource requirements on the host.

3.1.1 Benchmarking capacity of smartNICs

Task offloading needs to be done with careful considerations, lest it becomes the source of a bottleneck. As an illustration, computation cores on ASIC based NICs are generally non-preemptive, such that tasks allocated to a CPU core will run till completion. While this is not a cause for concern for packet processing tasks, which are generally small, offloading a long-running task on such a CPU core can have a detrimental effect on the NICs performance. On a similar front, the increased parallelism at the NIC comes at the cost of reduced computational power on a per-core basis. CPU intensive tasks thus might perform better at the host compared to offloading to the NIC. We intend to undertake benchmarking exercises to determine the performance of tasks chosen from a spectrum of resource requirement profiles when offloaded to the NIC, under different runtime conditions to define classes of workloads best suitable for offloading to the NIC.

3.1.2 Benchmarking workload performance under resource contentions

The presence of parallel cores allows us to offload multiple tasks concurrently on the NIC. These tasks will execute in the presence of their co-tenants as well as general packet processing workloads. This can lead to resource contentions and degraded performance for all or a subset of workloads. A thorough understanding of smartNIC performance is thus required to ensure optimum performance under resource contention and prevent bottlenecks in vital pipelines like the packet processing pipeline sharing the same NIC resources. In this regard, we intend to undertake performance benchmarks under different resource contention and co-tenant stress scenarios.

Determination of policies for optimum performance would require an understanding of the performance of tasks under different execution environments. We intend to exploit the performance benchmarks we described here to form policies described in the next section.

3.2 Prototyping a Distributed File System

Our objective is to identify the most promising distribution for data, metadata, and compute among the storage/compute technologies (DRAM, SSD, NVMM, smartNIC, and CPU) and the related communication techniques (RDMA and smartNIC) to design a prototype for a high-performance distributed file system. We discuss some interesting design options that leverage the new hardware and technologies for our first prototype. Our benchmarking exercise (§3.1) would answer some of the research questions that we ask in this section.

3.2.1 Leveraging smartNICs for performance acceleration

We have seen the networking hardware advancements such as programmable packet parser/pipeline, high-level of parallelism (200+ processing cores), dedicated accelerators, and onboard memory. Figure 3.1 shows a possible storage stack. We discuss several opportunities where smartNIC capabilities can be leveraged for performance acceleration.

Hierarchical caching

Modern smartNICs like Netronome Agilio [17], Broadcom Stingray [2] and Mellanox Bluefield [16] have considerable memory onboard (e.g., 4 GB). We can leverage the second layer

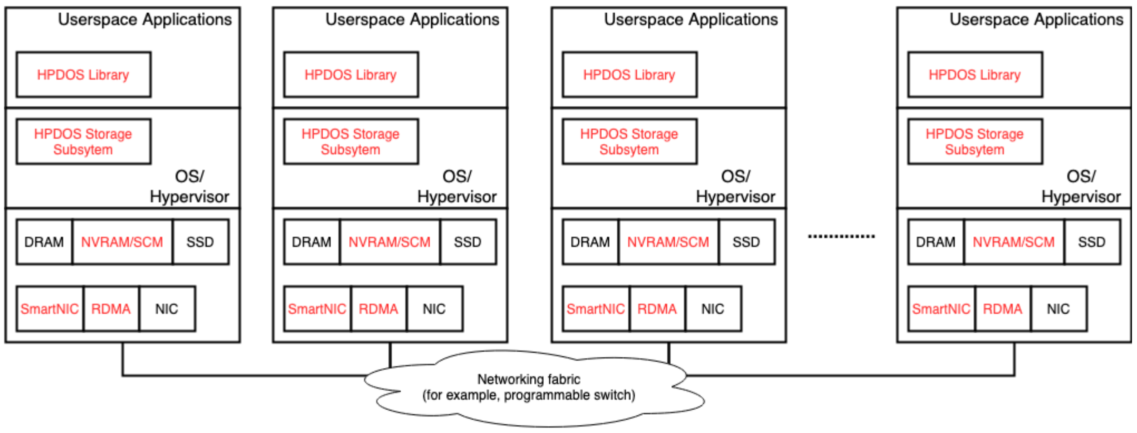


Figure 3.1: A SmartNIC enabled Distributed File System stack

of cache outside of the host's DRAM cache. The smartNIC-based cache can be leveraged on the server-side to process client requests independent of the host's involvement.

Another interesting question is, can we perform caching outside of the host and achieve performance beyond the maximum host bandwidth? Today's data centers use programmable switches as a connection fabric that have significant onboard memory which can be used by the offloaded applications. We can extend the throughput beyond the host's capacity by leveraging the Top-of-the-Rack (ToR) switch's memory as a cache (second or third layer) for frequently accessed (read heavy-hitters) file objects stored at the hosts within the same rack. However, the introduction of a cache involves developing suitable management policies to leverage the cache effectively.

- Maintenance of cache coherency between the host and smartNIC memories during concurrent read/write for the same file object.
- The limited size of the NIC onboard memory warrants the development of effective provisioning, ageing, and eviction policies for file objects (data or metadata).
- Under a hyperconverged compute-storage view, the reader and writer nodes are not separated; any compute unit can simultaneously play both the roles. While a writer or a reader would prefer the host DRAM cache for local file objects, a consumer with a remote request would prefer a remote smartNIC cache hit for latency purposes. This duality requires policy decisions to be in place to find the best location for file objects based on data access patterns.

Prefetching metadata and data

Local reads are typically much faster compared to remote reads even when zero-copy kernel bypass optimizations are involved [14]. Effective identification of an application's working set allows us to eagerly prefetch data from multiple read targets. This promises to improve read latency at the same time reducing network traffic by reducing effective remote reads. This can be supplemented by dynamic client side cache resizing based on the working set of applications using the cache. Another interesting design question is the dynamic sizing of cache space between metadata and data. Typically file/object systems have hard coded cache division, but we believe that this division should be dynamic - perhaps not real time but certainly periodically resized.

To understand the benefits and overheads of employment of prefetching, we want to answer the following questions.

- What type of objects should be prefetched? For example, data vs. metadata and small size vs. large size
- What is the destination of prefetch, smartNIC, DRAM, NVMM, or NVMe SSD? The incoming requests would prefer smartNIC whereas outgoing requests prefer DRAM cache. If the in-memory cache is unavailable, we can prefetch small objects to NVMM and the application can use the PMDK library (memory-mapped I/O) to access the cache, whereas prefetch large objects to NVMe SSD and the application accesses them using SPDK library (kernel bypass).
- Which replica should be used for prefetching? The prefetch load should not become the reason for a bottleneck. We require to establish policies to choose a source replica for prefetch such that the load is balanced among the replicas and the network links. Prefetching is best-effort, therefore, our policy should be designed to limit prefetch under high load conditions.
- Where do we run the prefetch logic, CPU, smartNIC, or ToR switch? We should identify the scenarios when offloading the prefetch logic to smartNIC or a ToR switch is beneficial.

Offloading storage solution components to smartNICs

Recent advancements have improved computation capabilities on smartNICs, ranging from ASIC based NPUs [17] to general purpose SoC multicore processors [2, 16]. This opens up avenues to offload storage computations (complete/partial) from the host CPU to the NIC. Offloading storage solution computations such as consistency and gossip protocols, replication and failure handling, and analytics engine to understand application storage to the smartNIC has a two-pronged advantage. First, making decisions at the network edge reduces latency, allowing nodes to react faster to changing system states. Second, this frees up the host CPUs to user-space applications that would otherwise have been tied down with storage management tasks.

One of the concerns related to computation offload is resource contention. In the case of a multi-tenant model, applications potentially from separate customers operate in isolated environments while sharing a pool of resources, which leads to scenarios where a certain class of resource-consuming applications can affect its co-tenants' performance. The smartNIC CPU and memory cache are examples of such resources. While there are battle-tested methods for partitioning host-side resources like virtual machines and containers, such hardware or software-assisted, partitioning mechanisms are not currently available for smartNIC resources. Developing such partitioning mechanisms would prevent resource consumption monopolization and enable provider-side QoS guarantees while protecting against adversarial workloads.

We want to answer a few questions to ensure efficient utilization of computing resources and improved compute performance.

- What application characteristics determine if the application is offloadable or not? For example, if most requests for an offloaded application require smartNIC processing as well as the user-space application processing, we should not offload such applications.

- How much application compute should be offloaded to the smartNIC? We can offload the entire application or partition the application functions and offload the functions that are most frequently used by remote requests. For example, in the case of replication, we offload replication logic to the CPU and the replication messages are handled by the SmartNICs, or we can process both at the smartNIC.

- Since the request characteristics are dynamic; the offload decisions should also be dynamic. What are the benefits of a dynamic offload decision? What should be the decision-making period? What are the overheads?

- If multiple host applications are offloaded to the smartNIC, how do we manage the resource provisioning and partitioning for each application?

3.2.2 Collaboration between resources

- A direction worth exploring is how we can leverage SmartNICs for metadata lookup, and at the same time leverage, one-sided RDMA for data fetch. The latter has been shown to be effective by the existing literature, but the open question is whether we can increase the effectiveness by complementing one-sided RDMA with metadata lookup offloaded to SmartNICs.

- The other interesting question is, can we offload transaction processing to the smartNIC?

  Consider an incoming client transaction request that involves multiple object lookups, where some objects are cached on the smartNIC, and others need to be remotely fetched. We could improve the performance if we can perform all the operations at the smartNIC. To achieve this, we should be able to offload the processing of transport protocols (RPC/gRPC) to the smartNIC, and the NIC could use RDMA to fetch the remote object into its cache.

- We have three resources for storage access, smartNIC, RDMA, and CPU with the recent hardware. We want to dynamically identify their capacities and then provide control knobs that dynamically provision work between the smartNIC, RDMA, and the CPU.

Bibliography

[1] Laurent Bessières et al. "Development of a probabilistic ocean modelling system based on NEMO 3.5: application at eddying resolution." In: *Geoscientific Model Development* 10.3 (2017).

[2] Broadcom. *Broadcom Stingray.* `https://www.broadcom.com/products/ethernet-connectivity/network-adapters/smartnic`. Accessed: 2020-10-24. 2020.

[3] Wei Cao et al. "PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database". In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1849–1862.

[4] Youmin Chen et al. "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020, pp. 1077–1091.

[5] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[6] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: `10.1145/1323293.1294281`. URL: `https://doi.org/10.1145/1323293.1294281`.

[7] Gert Everaert et al. "Risk assessment of microplastics in the ocean: Modelling approach and first conclusions". In: *Environmental Pollution* 242 (2018), pp. 1930–1938.

[8] Brad Fitzpatrick. "Distributed Caching with Memcached". In: *Linux J.* 2004.124 (Aug. 2004), p. 5. ISSN: 1075-3583.

[9] Fei Hu et al. "ClimateSpark: An in-memory distributed computing framework for big climate data analytics". In: *Computers & geosciences* 115 (2018), pp. 154–166.

[10] David Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing.* STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 654–663. ISBN:

0897918886. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660.

[11] Daehyeok Kim et al. "Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 297–312.

[12] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[13] Zhen Liang et al. "DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory". In: *Asian Conference on Supercomputing Frontiers*. Springer. 2020, pp. 40–54.

[14] Ming Liu et al. "Offloading distributed applications onto smartNICs using iPipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[15] Teng Ma et al. "AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 757–773.

[16] Mellanox. *Mellanox Bluefield*. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet. Accessed: 2020-10-24. 2020.

[17] Netronome. *Netronome Agilio*. https://www.netronome.com/products/smartnic/overview/. Accessed: 2020-10-24. 2020.

[18] John Ousterhout et al. "The RAMCloud storage system". In: *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015), pp. 1–55.

[19] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[20] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 873–880.

[21] *Redis*. https://redis.io/.

[22] Antony I. T. Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Middleware '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 329–350. ISBN: 3540428003.

[23] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts: Seventh Edition*. "McGraw-Hill", 2019.

[24] Michael D Smith. "Overcoming the challenges to feedback-directed optimization (keynote talk)". In: *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. 2000, pp. 1–11.

[25] Ion Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: `10.1145/964723.383071`. URL: `https://doi.org/10.1145/964723.383071`.

[26] Peter E Strazdins et al. "Scientific application performance on hpc, private and public cloud resources: A case study using climate, cardiac model codes and the npb benchmark suite". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 1416–1424.

[27] Agnès Tellez-Arenas et al. "Scalable interactive platform for geographic evaluation of sea-level rise impact combining high-performance computing and WebGIS Client". In: *Communicating Climate Change Information for Decision-Making*. Springer, 2018, pp. 163–175.

[28] Jean-André Vital et al. "High-performance computing for climate change impact studies with the Pasture Simulation model". In: *Computers and electronics in agriculture* 98 (2013), pp. 131–135.

[29] Jian Yang, Joseph Izraelevitz, and Steven Swanson. "FileMR: Rethinking {RDMA} Networking for Scalable Persistent Memory". In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 2020, pp. 111–125.

[30] Jian Yang, Joseph Izraelevitz, and Steven Swanson. "Orion: A distributed file system for non-volatile main memory and RDMA-capable networks". In: *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 2019, pp. 221–234.