Tromos : a software development kit for virtual storage systems

Fotios Nikolaidis

université PARIS-SACLAY

NNT : 2019SACLV033

Thèse de doctorat

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Tromos: A Software Development Kit for virtual storage systems

Thèse de doctorat de l'Université Paris-Saclay
préparée à Université de Versailles-Saint-Quentin-en-Yvelines

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Versailles, le 22/May/2019, par

M. FOTIOS NIKOLAIDIS

Composition du Jury :

Mickael Krajecki Professeur, Université de Reims	Présidente, Rapporteur
William Jalby Professeur, Université de Versailles	Directeur de thèse
Soraya Zertal MCF/HDR, Université de Versailles	co-encadrant de thèse
Philippe Deniel CEA/DAM, Bruyere le chatel	co-encadrant de thèse
Maria Perez Professeur, Université de Madrid	Rapporteur
Jacques Jorda MCF/HDR, Université de Toulouse	Examinateur
Denis Barthou Professeur, Université de Bordeaux	Examinateur
Gael Thomas Professeur, Telecom Sud Paris	Examinateur

Titre: Tromos: un cadre pour la construction de systèmes de stockage distribués

Mots clés: Blob storage, Stockage distribué, Stockage sur le nuage, Storage containers

Résumé: Parmi la mise à l'échelé, la portabilité, la distribution du charge de travail, et autres raisons, les applications vont bénéficier la plateforme du stockage des données qui present le code le plus efficace. Cela, cependant, n'est pas trivial. Chaque plateforme aboutie aux accomplissements diversifiés and, par conséquent, elle requiert le rôle d'une application pour achever plusieurs chemins de codage. L'implémentation des tels chemins n'est pas évidente. Elle requiert une grand effort, des hautes compétences de programmation et suivant elle contient un fort risque des erreurs. Les problèmes se présentent quand les applications doivent de supporter plusieurs stockages des données en parallèle. Dans cette dissertation, on introduit le terme "storage containers" comme une évolution naturelle du stockage des donées. Les "storage containers" sont des application intermédiaires de gestion des données qui sépare la logique d'I/O de la logique d'affaires et la logique de computations d'une application. Dans cette thèse, nous introduisons le terme « récipients de stockage » comme la prochaine logique dans l'évolution du stockage. Il s'agit d'un logiciel intermédiaire de gestion des données qui sépare la logique d'I/O de la logique métier et de calcul d'une application. Autrement dit, ils séparent les changements apportés au code des applications par les utilisateurs scientifiques des changements apportés aux actions d'I/O par les développeurs ou les administrateurs. Contrairement aux systèmes « à usage général » existants, dont l'objectif est d'effectuer « décemment » pour le plus grand nombre possible d'applications, les systèmes « application-tailored » conservent la connaissance et ne fonctionnent de façon optimale que pour une seule application. Une telle conception aide à reporter les décisions d'I/O jusqu'à la phase de déploiement, qui est la clé de la Transférabilité ; de la sécurité de l'autorité la moins élevée ; de la fédération des fournisseurs de stockage non-collaboratif; de l'échelle d'application de deuxième ordre indépendante au stockage de données ; et la distribution de données des règles avec des critères comme la résilience, le rendement, l'efficacité du stockage et le coût. Le fait qu'un tel logiciel intermédiaire soit « personnalisé » à l'application, signifie également que chaque application doit mettre en œuvre sa "saveur". Pour ce faire, nous introduisons un SDK, appelé Tromos, pour les développeurs, afin de synthétiser des systèmes de stockage distribués personnalisés sans programmation des systèmes. Tout ce qu'il faut aux développeurs est de modéliser l'environnement souhaité dans un fichier de définition. Le SDK va gérer le reste du processus. Il est distribué avec une large gamme de plugins intégrés pour le traitement des I/O, le traitement des requêtes, les algorithmes de sélection, les méthodes de reconstruction de données, le traitement de cohérence, et la visualisation. Pour faire une analogie, "storage containers" découplent la gestion des données de la plateforme de stockage physique de la même manière que les containers Docker découplent l'environnement d'application des serveurs physiques. En guise de démonstration de faisabilité, nous utilisons Tromos pour prototyper des environnements de stockage personnalisés que nous comparons à Gluster, un système à usage général appartenant à RedHat. Les résultats ont montré que les environnements auto-produits surpassent Gluster simplement en enlevant toute fonctionnalité inutile de la ligne de stockage.

Title: Tromos: A Software Development Kit for virtual storage systems

Keywords: Blob storage, Distributed storage, Cloud Storage, Storage containers

Abstract:
Modern applications tend to diverge both in the I/O profile and storage requirements. Matching a scientific or commercial application with a general-purpose system will most likely yield suboptimal performance. Even in the presence of “purpose-specific” systems, applications with multiple classes of workloads are still in need to disseminate the workload to the right system. This strategy, however, is not trivial as different platforms aim at diversified goals and therefore require the application to incorporate multiple codepaths. Implementing such codepaths is non-trivial, requires a lot of effort and programming skills, and is error-prone. The hurdles are getting worse when applications need to leverage multiple data-stores in parallel. In this dissertation, we introduce “storage containers" as the next logical in the storage evolution. A “storage container" is virtual infrastructure that decouples the application from the underlying data-stores in the same way Docker decouples the application runtime from the physical servers. In other words, it is middleware that separate changes made to application codes by science users from changes made to I/O actions by developers or administrators.
To facilitate the development and deployment of a “storage container" we introduce a framework called *Tromos*. Through its lens, all that it takes for an application architect to spin-up a custom storage solution is to model the target environment into a definition file and let the framework handles the rest. *Tromos* comes with a repository of plugins which the architect can choose as to optimize the container for the application at hand. Available options include data transformations, data placement policies, data reconstruction methods, namespace management, and on-demand consistency handling.
As a proof-of-concept we use *Tromos* to prototype customized storage environments which we compare against Gluster; a “general-purpose” system owned by RedHat. The results showed that the auto-produced environments outperform the more mature Gluster by merely removing the unnecessary overhead of unused features.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l’Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

One picture, a thousand thanks !!!!

This dissertation is dedicate to our friend Dimitris Ganosis

Contents

List of Tables

List of Figures

‘

Chapter 1

Introduction

In this Chapter we present the evolution of the various persistent storage interfaces that are broadly available nowadays. During this historical recursion, we are introducing the reader to the purposes these interfaces were trying to fulfill, how they were designed according to the technology of the time, and how those decisions affect today's applications.

1.1 Block Devices

In the dawn of computing, the dominant media for secondary storage were tape drives and hard disk drives. Both are rotating devices that use electromagnetic heads to store and retrieve data from surfaces coated with magnetic material. Alternatively referred to as external memory, secondary memory, and auxiliary storage, a secondary storage device is a non-volatile device about two orders of magnitude cheaper than the primary storage.

Device controllers use the idea of logical block addressing (LBA) to abstract the direct addressing of hardware geometry (e.g., platters, cylinders, and tracks) into evenly-sized blocks of raw data. The physical block size -usually 512 bytes- corresponds to a disk sector, which is the smallest unit of data that the disk controller can read or write in a single operation. During an I/O operation, the physical media is continuously rotating. However, the rate at which data are written or read to the drive is not deterministic; it depends on the rate at which data is supplied or demanded by its host. If the host rate is higher than the device rate, the device will become a bottleneck and can potentially start rejecting data. If the host rate is lower than the device rate, the device will spin, wasting energy and blocking others clients from using it. The "hardware" solution to cope with this difference is to physically slow-down or speed-up the drive according to the host rate. The elevator algorithm (also SCAN) is a disk scheduling algorithm that reorders requests in such a manner to minimize the motion of the disk's arm and head; thus making access faster and more efficient. Bear in mind that for non-rotating disks like Solid-State Drives (SSD) and the newer non-volatile memories (NVM) the location of the data does not affect performance.

Typically, block devices read and/or write in full blocks; if a write operation does not fill up a full block, the user needs to read the remaining portion from the current contents of the file,

merge them, compute the updated block and, finally, pushing back to the device the updated block. UNIX kernel uses logical blocks, or clusters of physical blocks, to transfer data from and to devices. The logical block size is set to the page size of the system by default - 4k bytes. A page is the smallest unit of data for memory management in a virtual memory operating system that represents a fixed-length contiguous block of virtual memory, described by a single entry. Logical block size imposes a time-space tradeoff. Much of the delay in a rotational disk is due to the time it takes to correctly position the read/write heads above the disk platters. Larger blocks pack more data in a contiguous space and yield higher sequential throughput than several smaller transfers. Oppositely, smaller blocks allow for more I/O operations per second (IOPS).

With the term “physical volume” we refer collectively to a physical drive, or a hardly provisioned portion of it called partition. A logical volume is a “virtual” storage device which can span multiple physical volumes (e.g., Redundant Array of Independent Disks). Thereby, logical volumes are composable by “block layers“, with each layer performing data transformations in a non-disruptive way to the upper layers. “Block devices” oppose to “character devices” that handle data as streams of bytes (e.g., video or audio devices).

Storage Area Network A storage area network (SAN) is a dedicated high-speed network of interconnected storage servers presents as shared pools of devices. In general, SAN combines storage servers in master/slave high-availability configurations where the slave needs to be ready to take over the master. Clients request access to the devices by sending out SCSI commands to the SAN. Fibre Channel (FC) SANs have the reputation of being an expensive and local network that yield low-latency and high throughput. iSCSI addresses cost and locality issues by encapsulating SCSI commands into IP packets that do not require an FC connection. SAN has been successfully running mission-critical applications like Oracle, SAP, Microsoft Exchange, and Microsoft SharePoint. SAN usually do not scale over 64 or 128 nodes.

1.2 Filesystems

Logical block addressing was undoubtedly an advancement over geometry addressing. Block devices, however, suffered from lack of organization. They were simply receiving commands to store that block, or load the other block, with no understanding of the content. It was up to the programmers to ask the right block. Filesystems emerged as mean to organize data into files and file hierarchies. In principle, a filesystem deals with two distinct aspects i) how to present the organizational structure to the end-user 2) how to map logical files into physical devices. It does so by using structures residing both in-memory and on-disk.

The superblock structure provides the operating system with high-level information for mounting the filesystem, e.g., the total number of blocks, free blocks, root index node. Filesystems use inodes (short for index node) to maintain metadata information about files, directories, or symbolic links. File-level and directory-level system calls map directly to inodes, e.g., open(), read(),

write(), flush() for files, create(), lookup(), link(), mkdir() for directories. Inodes that represents files also maintain a table pointing to the written blocks and a virtual address space of fixed-sized pages. A page is the memory-equivalent of a block on a block device. Dentries track the relationship of one entry to other entries in the file system as well as physical data (such as the file name). A file system has one root dentry without a parent, whereas all other dentries have parents, and some have children. Dentries are in-memory structures constructed by on-disk inodes.

File Handler As users create or expand files, the kernel allocates disk space in full logical blocks or segments of logical blocks called fragments (e.g., 4KB block; 1KB fragment). When files need extra disk space, the kernel first allocates full blocks, and then it allocates one or more fragments of the block for the data. For every opened file, the operating system maintains a handler for keeping the state of a file. They act as middlemen for translating memory pages to the storage blocks. When a process modifies a page (an offset in the file), the handler marks the page as dirty. Dirty means that the data exist in the page cache but is not persisted on the storage device yet. That gives time to the operating system to aggregate data into contiguous blocks so to reduce fragmentation of disk space that results from unused holes in blocks.

1.2.1 VFS

Institute of Electrical and Electronics Engineers (IEEE) specified in the 1980s Portable Operating System Interface (POSIX) application programming interfaces (API) for applications to interoperable across System V and BSD Unix operating systems. Virtual file system switch (VFS) is a POSIX kernel abstraction for mounting several types of file systems into the same tree structure while retaining a uniform way for users to navigate file system hierarchies. It encompasses most of the system calls programmers are familiar with, e.g., open(), close(), read(), write(), and lseek(). VFS applies generic filesystem actions and vectors requests to the correct filesystem for further processing. Filesystems may store data on local devices, remote devices, or even in "virtual" processes in userspace (Figure 1.1).

Generic Block Layer

VFS access block devices through Generic Block Layer (GBL) abstraction. It is oblivious to the specific device implementation. GBL abstracts physical devices (e.g., HDDs, SSDs) or "virtual" devices run as userspace process (e.g., Network Block Device in Linux). Figure 1.2 presents how GBL connects with VFS and the block device drivers.

In Userspace

Filesystems that reside on kernel run with supervisor privileges and can crash the entire machine if they experience an error. Some programs user their custom VFS implementation (e.g., GnomeVFS for Gnome desktop) to avoid the kernel burdens. Especially in High-Performance computing (HPC) environment is it common to implement userspace filesystems as preloaded libraries. The drawback is that such filesystem is not accessible by third-party processes; a process cannot do “ls” inside a GnomeVFS tree. FUSE is a kernel protocol that bridges VFS with userspace filesystems. That separation makes it much more sensible to write filesystems with lots of external dependencies.

Over the network

A clustered file system is a file system which is shared by being simultaneously mounted on multiple servers. Clustered file systems can provide features like location-independent addressing and redundancy which improve reliability or reduce the complexity of the other parts of the cluster. VFS layer hides the network communication from client programs on clustered nodes. Clients can access the filesystem like any other local filesystem.

Shared-disk is a type of clustered filesystem that allows multiple clients to gain disk-level on NAS. It employs fencing mechanisms for concurrency control that allows the nodes to fail without unintended data loss and without affecting the access of other nodes. The most prominent filesystems of this type include Oracle Cluster File System (OCFS), IBM General Parallel File System (GPFS), StorNext from Quantum, Veritas from Symantec, and VMFS from VMware/EMC Corporation.

Distributed file systems do not share block-level access to the same storage but use a file-level protocol. Network File System (NFS) protocol, the de facto distributed file system in Linux, builds on the Open Network Computing Remote Procedure Call (ONC RPC) system. A Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network.

1.2.2 POSIX limitations

Like most of the designs in the bronze era of computing, POSIX conceived to favor preciseness and correctness. It relies on stateful handlers and strong event ordering. Unfortunately, POSIX design does not reflect today’s requirements for concurrency, scalability, and real-time. Preciseness and correctness are not that much of an issue any longer; No one would trade extra latency waiting for a Google search to load all the possible results before data become accessible. Next, we present an extensive, but not exhaustive, list of facts that render POSIX programming obsolete for today’s standards.

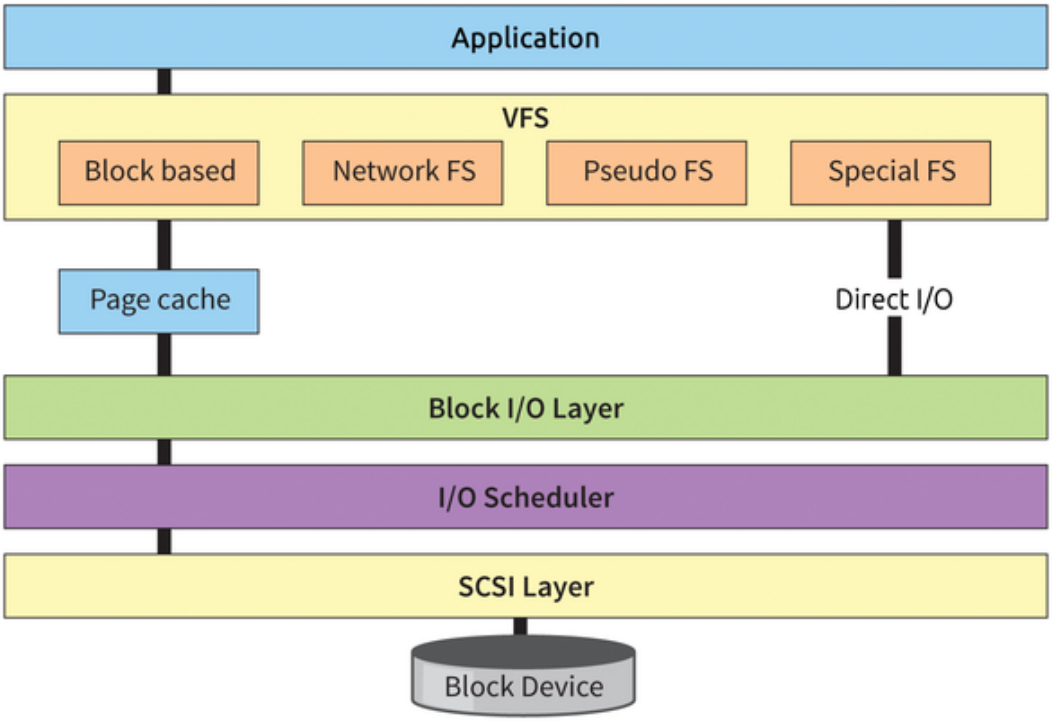

Figure 1.1 – The Linux I/O stack: Different layers provide a single, virtual interface to the respective layers above, thus hiding the complexity and peculiarities of their subordinate world (Source [20])

Deprecated API

POSIX was designed for single hosts, with a single CPU core, for storing data to rotating devices. Parallel processes at that time were not physical but “scheduled”. Operating systems realized pseudo-parallelism by occasionally switching CPU execution from one task to another. That required stateful tasks (or handlers) so to resume later back to the point they were switched (e.g., write offset). Most developers have already encountered seek() function at least once. Its original purpose was to position the magnetic head before a data transfer. Systems with stateful handlers may scale to hundreds, or even thousands, of processes trying to perform I/O, but cannot scale to millions or billions of them. Further, most of the modern data are "units" that do not require much random I/O (e.g., images, music, videos).

Another characteristic of rotating devices is that they can serve only one task at a time; they are single-channeled. Hence, all I/O requests going through the kernel which was deciding how and when to flush data to devices. Given that at the time the bottleneck was on the I/O from kernel to devices, the overhead to perform a system call and copying data user space to kernel space was negligible. None of them is no longer valid. Modern storage devices such as NAND flash-based solid state drives (SSDs) and non-volatile memories (NVM) exhibit high-throughput, low-latency on random I/O and a high degree of parallelism. Along with memory-access technologies such as zero-copy and RDMA, have moved the bottleneck from the devices to the I/O path itself. Hence, prolonged and obscure I/O stack, with a single point of congestion

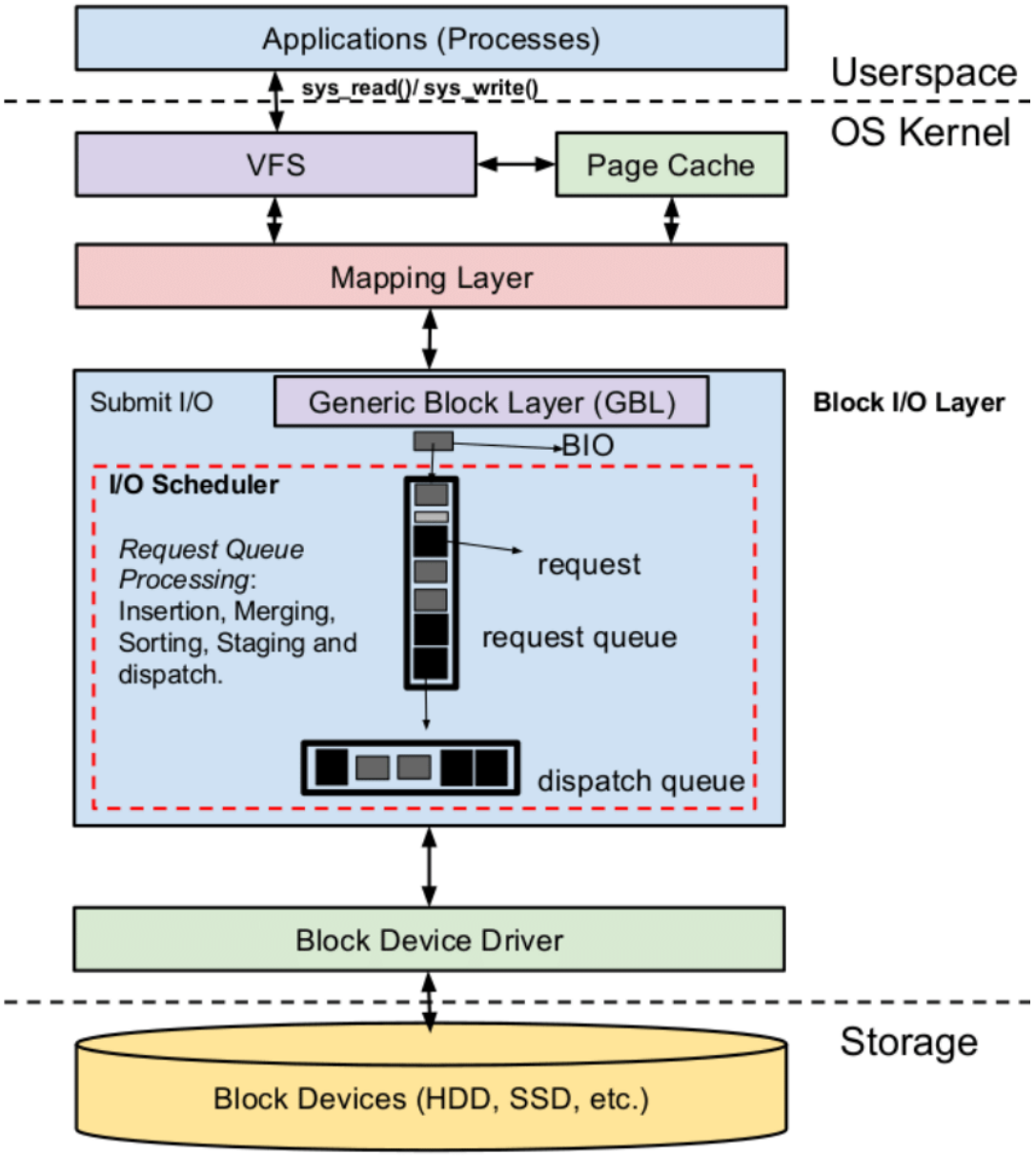

Figure 1.2 – Architecture of Linux Kernel I/O stack (Source [85])

(e.g., VFS) and without support for newcomer technologies are not good candidates for today's versatile storage environment.

POSIX organizes files as directories in a single namespace. For the filesystem implementation, it means that i) coordination is needed even for files under different directories since they belong to the same namespace. ii) it must enforce a mechanism for access control on top of the namespace. Access control is not a synonym to isolation and usually enforces unnecessary overheads for use in HPC computing. For example, the MPI-IO standard does not expose the file hierarchy or permissions to the end user. Therefore, applications leveraging these libraries do neither need nor expect these features. It is also common for many application to keep organization externally (e.g., on a database) and use filesystems only to dump data content.

Strong Semantics

POSIX mandate changes to a shared file to become immediately visible to all the processes; enforces strong consistency semantics. For example, a write() will block application execution until the system can guarantee that any other read() call will see the latest written data. For a host with a single CPU core, it is "easy" to guarantee immediate visibility since all processes were running on the same core. That, however, is not possible on today's multicore architectures as processes may run on different cores. Synchronization requires communication and the speed of light bounds communication. Thereby, fine-grained locks are impractical. Coarser locks are needed, which increase the locked region, which hinders potential parallelization gains.

According to Amdahl's law, the performance gains from parallel task execution are limited proportionally the sequentially executed parts. The formula 1.1 gives the expected theoretical speedup in latency of the execution of a task at the fixed workload of a system whose resources are improved. That metrics gives the "scalability factor". If loose 5% of a processor power every time a CPU is added to the system, then the "scalability factor" is 0.95. A scalability factor of 0.9 means that only 90% of the resource will be usable.

$$speedup = (s + p)/s + p/N \tag{1.1}$$

Local I/O stack

There can be multiple intermediate processing layers between applications and storage devices. Such layer may recursively flush until the request goes to the device, or buffer the data and immediately acknowledge the I/O completion. Data residing in the buffer migrate to next layers in a background (asynchronous) task, either explicitly to the user (by calling sync()), or periodically by the system. As long as requests traverse the same I/O stack, it is easy for the system to serve next read() directly from the buffered or cached data. No need to go to the devices.

The first associated risk is that in case of a crash the data are gone. POSIX partially "protects' filesystem structure by first modifying the metadata, and then the data. Even if the payload gets corrupted, the filesystem structure will remain consistent. If the file shrinks, the unmodified data will still exist on the disk but without being accessible from the filesystem. If the file grows, subsequent requests will see the correct metadata but corrupted data. (Note 1: yet another example of POSIX over-engineered API. Note 2: Journalling is not an integral part of POSIX but uses its calls).

The second pitfall is that modern storage requirements go far beyond the capabilities of a single node. Instead, many nodes across a network contribute their resources to form a distributed system. Hence, clients on difference nodes traverse different I/O stacks. The solution is to apply coarse grain locks using Distributed Locking Mechanisms (DLM), which against enforces high synchronization overheads.

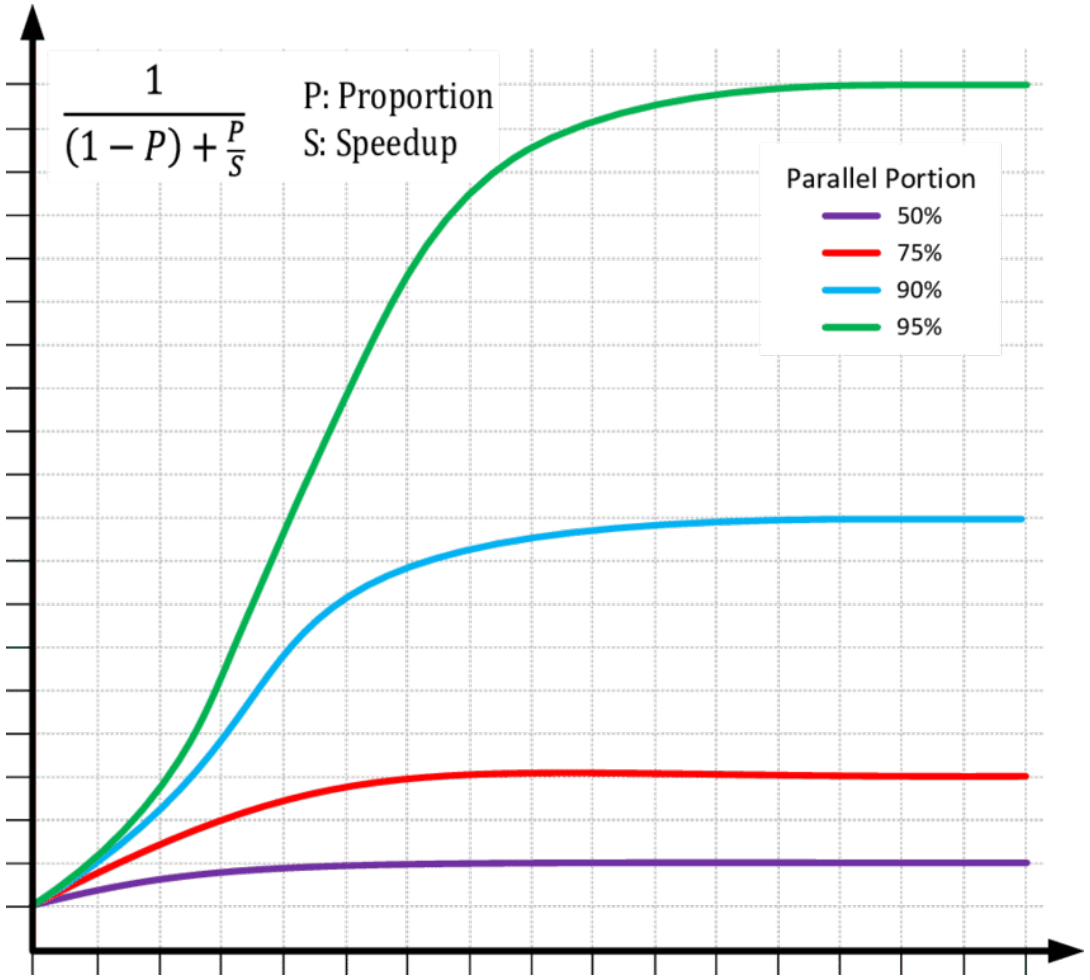

Figure 1.3 – Amdahl's law: the speed up for an application is reversely proportional to its sequentially executed code

1.3 Object-based Storage

Files in Filesystems are logical mappings pointing to physical data location. They include little to no contextual knowledge (metadata) as to what these data are. Object storage architectures organize data determined by the user to be logically related to self-described units. Objects carry information across layers as an indivisible unit that encapsulates raw data, user-expandable metadata, system-handling attributes, and a globally unique identifier. When an object passes through a layer in the I/O stack, the layer determines how to handle the object based on the values in the attributes that it understands. All other attributes pass along unmodified. Hence, the system handles objects marked for efficient distribution and parallel access different from objects marked as temporary.

The unique identifier algorithmically encodes the physical location of the object. In the case of consistent hashing [62], storage nodes are assigned a range of keys within a hash space, covering the entire range with no holes or overlaps. Object identifiers are keys in that space. The node that serves the closest range of keys to the object identifier is the one that holds the object.

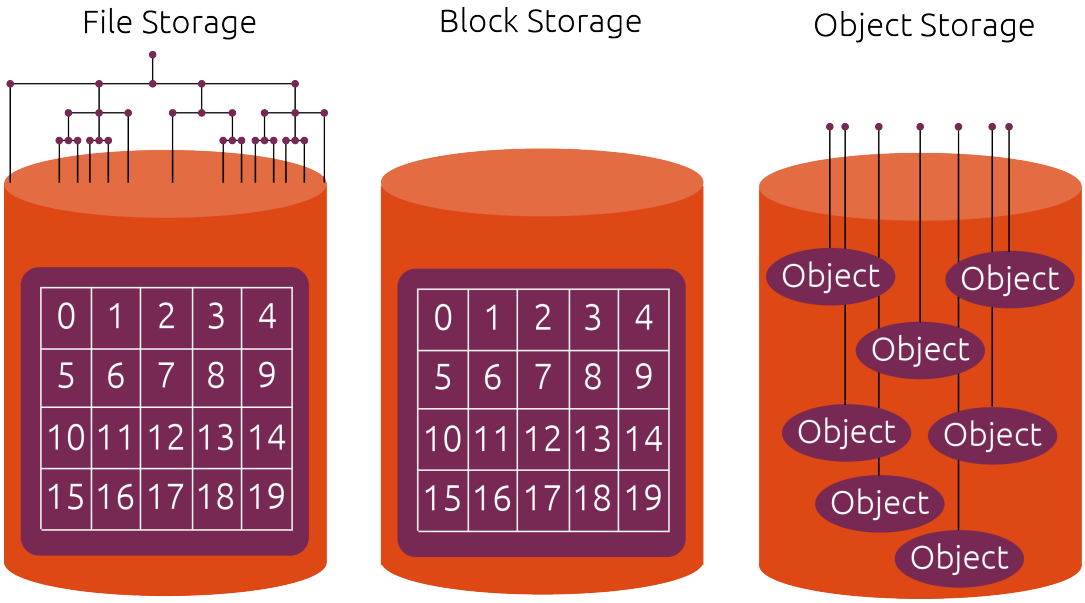

Figure 1.4 – Organizational differences among filesystems, block devices, and object storage (source [24])

Such a method eliminates the need to ask a lookup and traverse long hierarchical structures. Content-addressable storage (CAS) [123] is a way to store an object based on the content, not on the given name. When CAS systems store a datum, they hash the content and generate a unique identifier for it. Since identifiers derive from the content, any change to a data element will necessarily change its content address. Hence, its usage is common for high-speed storage and retrieval of fixed content, such as documents stored for compliance with government regulations.

Objects are immutable that do not provide the ability to edit just one part of the payload. In object-based storage, modifying a file means that users have to upload a new revision of the entire file. That can significantly impact performance if modifications are frequent. high-activity IO operations such as caching, database operations, or log files, should not use object-based mechanisms. Block storage mechanisms are better suited to these activities. Being of fixed-sized and relatively small, they can be accessed, modified, and stored individually, thus enabling efficient random I/O access.

For object-based systems, objects are keys in a flat namespace with no hierarchy or relations between them. Even though objects are not stored “together” on the same physical medium, application developers can still retain the relationship between them. Plus, the object storage model allows users to search through the metadata a great deal faster than compared to the block-based approach. A good use case to exemplify that is big data analytics: When dealing with increasingly large number of files, the object storage method of indexing the data and location (the metadata), and thus fixes the main problem connected with big data, which is scalability. Objects are a better fit for static and metadata-rich content such as medical imaging, data backups and archival images, and multimedia files (videos, pictures, or music).

Although object-storage became the flagship for Cloud Storage, its origin predates by at least

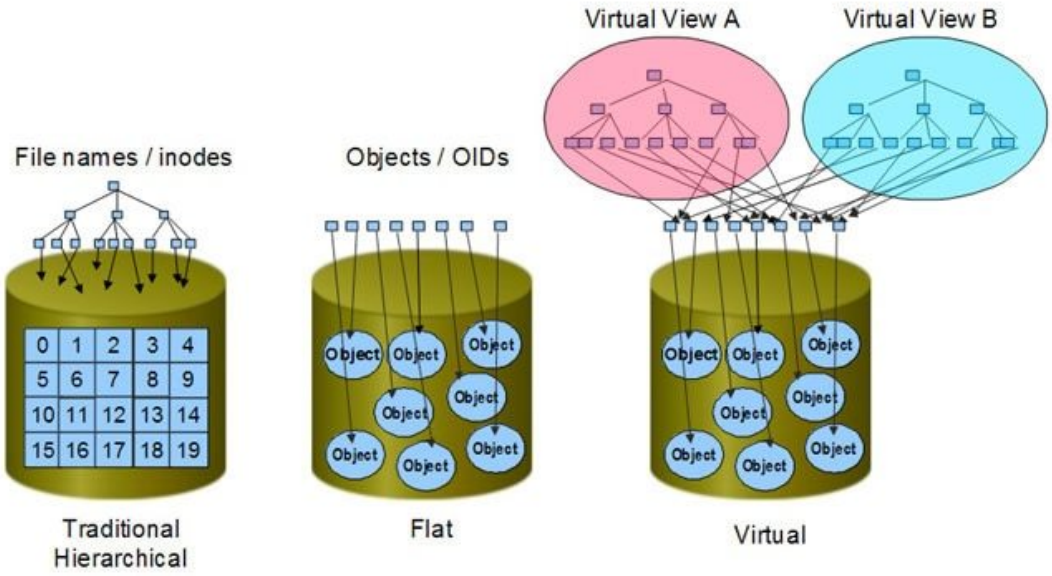

Figure 1.5 – Three approach for namespace management: hierarchical, flat (none), application-specific

a decade. In 1994 a workgroup initiated by Carnegie Mellon, and continued by SNIA, standardized a set of ANSI T10 SCSI commands for Object storage devices (OSD) [105]; an intelligent evolution of disk drives that handle and store objects instead of just placing data on tracks and sectors. OSD defines four different types of objects:

- The root object – The OSD itself
- User objects – Created by SCSI commands from the application or client
- Collection objects – A group of user objects, such as all .mp3 objects or all objects belonging to a project
- Partition objects – Containers for user objects and collections that share common security and space management characteristics, such as quotas and keys

The most known of such devices is Kinetic [108] from Seagate. The idea was that the drives did not use traditional block-based storage protocols like SAS and SATA, but instead stored objects written and retrieved over Ethernet. Effectively, each drive is a key-value store that manages its content.

1.4 Data stores

A data store is dedicated management software for storing data structures onto the storage media on behalf of the applications. three factors commonly drive their design

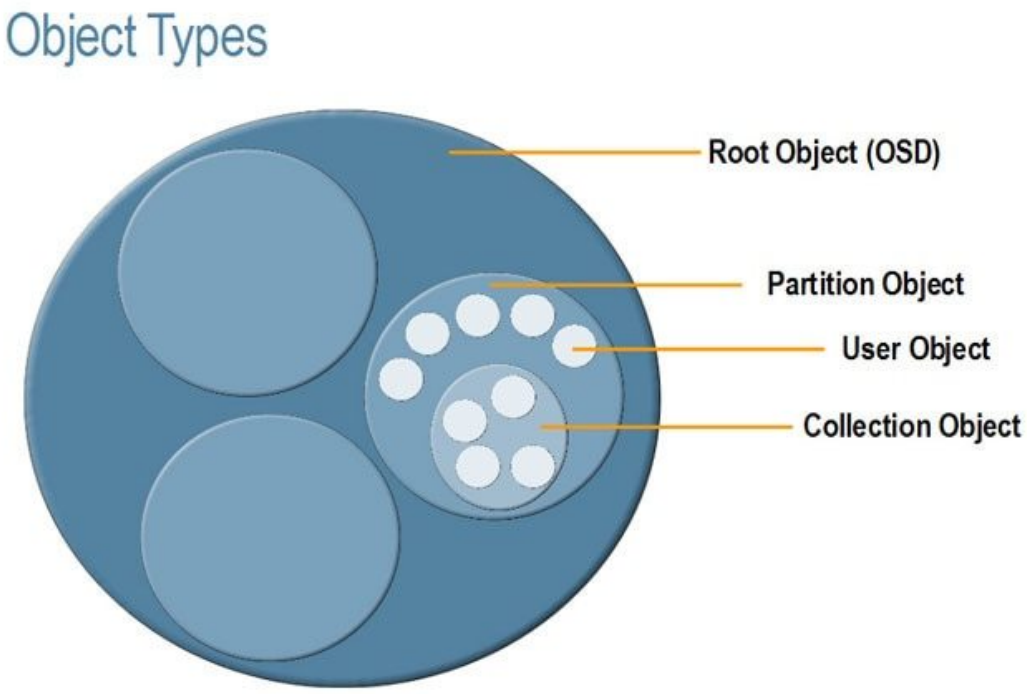

Figure 1.6 – Object types

- leverage infrastructure technologies concerning new storage media (e.g, non-volatile memories), advanced node interconnection (e.g., RDMA), host-scoped optimization (e.g., accelerators, zero-copy)
- provide specialized data-management (e.g., concurrency, replication, performance, encryption, storage-efficiency)
- provide domain-specific interfaces for accessing data (e.g., relations for structured data, key-value pairs for unstructured data, matrices for numerical data)

Because object-based storage architectures can scale out by merely adding nodes, many data-stores have adopted objects for their backend, as a solution to the increasing problems data growth and parallel access.

1.4.1 Cluster

A cluster is a group of tightly coupled and almost identical machines, connected by fast-link local area network, that collectively comprises a single powerful machine. To do so, it needs to orchestrate all nodes to work together and provide consistency of things such as caches and memory.

A parallel file system (PFS) is a type of clustered file system that distributes file data across multiple storage servers in order to provide concurrent access by multiple tasks of a parallel application. PFS use bulk object-based storage to store file data, file metadata, directory metadata,

directory entries, or symbolic links. Typically, they separate data servers from metadata servers, with the latter being involved only at the beginning of the operation to retrieve the file layout. The layout for each file defines the set of objects that will be used to hold the file's data. Data may be written to the objects in a round-robin manner (stripping) in order to avoid the bottleneck of a single server, or replicate to multiple servers for resiliency against crashes.

Typical examples include OrangeFS [94], Lustre [19]. Caring about concurrency and scalability these systems may feature relaxed semantics compared to POSIX. For example, pNFS [38] adheres to close-to-open consistency model that departs POSIX semantics for atomic write()s that become immediately visible to all clients.

1.4.2 Grid

A grid is an alliance of loosely coupled machines with possibly very different hardware configurations which work together to solve a given problem/crunch data. The fundamental difference between a grid and a cluster is that nodes in a grid are relatively independent and geographically distributed problems are solved in a divide and conquer fashion.

1.4.3 Cloud

Cloud storage is a model of computer data storage in which the digital data is stored in logical pools, owned and managed by the hosting vendor. Cloud vendors are responsible for keeping the data available and accessible, and the physical environment protected and running. Users lease storage capacity from vendors to store user, organization, or application data.

The difference between a cloud and a grid summarizes to [9]:

- Resource distribution: Cloud computing is a centralized model whereas grid computing is a decentralized model where the computation could occur over many administrative domains.
- Ownership: A grid is a collection of computers which is owned by multiple parties in multiple locations and connected so that users can share the combined power of resources.

Below we present a short of Cloud storage vendors that provide worldwide access to the objects through REST application programming interface (API) or client libraries.

- **Amazon S3**: Amazon S3 stores data as objects within resources called "buckets". AWS S3 offers features like 99.999999999 durability, cross-region replication, event notifications, versioning, encryption, and flexible storage options (redundant and standard).
- **Azure Blob Storage**: For users with large amounts of unstructured data to store in the cloud, Blob storage offers a cost-effective and scalable solution. Every blob is organized into a container with up to a 500 TB storage account capacity limit.

- **Google cloud storage**: Cloud Storage allows users to store data in Google's cloud. Google Cloud Storage supports individual objects that are terabytes in size. It also supports a large number of buckets per account. Google Cloud Storage provides strong read-after-write consistency for all upload and delete operations.

Below we present a short list of block-based Cloud storage. Block devices are typically accessible only from collocated cloud services (e.g., CLI).

- **AWS Elastic Block Storage (EBS)**: Amazon EBS provides raw storage – just like a hard disk – which users can attach to your ec2 instances. Once attached, users create a file system and get immediate access to the storage. Users can create EBS General Purpose (SSD) and Provisioned IOPS (SSD) volumes up to 16 TB in size, and slower, legacy magnetic volumes.
- **Rackspace Cloud Block Storage**: Rackspace provides raw storage devices capable of delivering super-fast 10GbE internal connections.
- **Azure Premium Storage**: Premium Storage delivers high-performance, low-latency disk support for I/O intensive workloads running on Azure Virtual Machines. Volumes allow up to 32 TB of storage.
- **Google Persistent Disks**: Compute Engine Persistent Disks provide network-attached block storage, much like high speed and highly reliable SAN, for Compute Engine instances. Users can remove a disk from one server and attach it to another server, or attach one volume to multiple nodes in read-only mode. Two types of block storage are available: Standard Persistent Disk and Solid-State Persistent Disks.

The best type of storage is application-specific as it has to balance the requirements for performance, resiliency, scalability, cost, consistency, and other factors. For example, object storage performs optimally for large data that follow write-once-read-many patterns; block storage performs optimally for small data that are frequently modified. Files are organizational abstractions atop objects, blocks, or records.

1.5 Designing scalable data-stores

Vendors protect user data against failures by storing multiple copies of the objects across their realm, which may span multiple data-centers. One way to achieve replication is by strongly-consistent operations. The clients must block waiting for the system to write all the replicas successfully. Strong consistency is a requirement for "real-time" systems where a read request must return the most updated version of the data. The tradeoff for the provided "correctness' is limited scalability and reduced availability as a result of hardware failures.

Vendors may postpone transfers across-data centers until a justified amount of data balance the transfer cost. That means for the client that must block for an arbitrarily long time. In practice,

the preferred solution for scalable systems is eventual consistency. The downside to eventual consistency is that there is no guarantee that a read request returns the most recent version of the data; different clients may see a different version.

Below we present the theoretical background to understand the above statements.

1.5.1 CAP theorem

Eric Brewer made the conjecture that there are three essential system requirements necessary for the successful design, implementation, and deployment of applications in distributed computing systems.

- **Consistency**. The client perceives that a set of operations has occurred all at once, i.e., data moves from one correct state to another correct state, with no possibility that readers could view different values
- **Availability**. The system remains operational 100% of the time, and every operation must terminate in an intended response. The key word here is every. Every node (on either side of a network partition) must be able to respond in a reasonable amount of time.
- **Network Partition**, or brain split. The system still needs to work even when some nodes in the system are unable to communicate with other nodes in the system.

An intuitive interpretation of the Conjecture is the following: in a distributed system, if some servers cannot access each other, either the distributed system will be unable to process some requests (lack of availability), or it will not behave like a single server (lack of consistency).

The typical "2 of 3" perception of CAP is misleading on several fronts [21]. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. In other words, all three properties are more continuous than binary. Availability is continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists. The availability of any system is the product of the availability of the components required for operation.

The last part of that statement is the most important since components that may be used by the system, but are not involved in the operation, do not reduce the perceived system availability. For example, when systems designers shard (partition) data it is highly likely that each shard can ensure complete consistency and availability during a partition.

1.5.2 BASE model

The BASE is an optimistic consistency model that accepts the data store state to be in flux. It favors immediate response than immediately consistent state or accuracy on the data reply. The

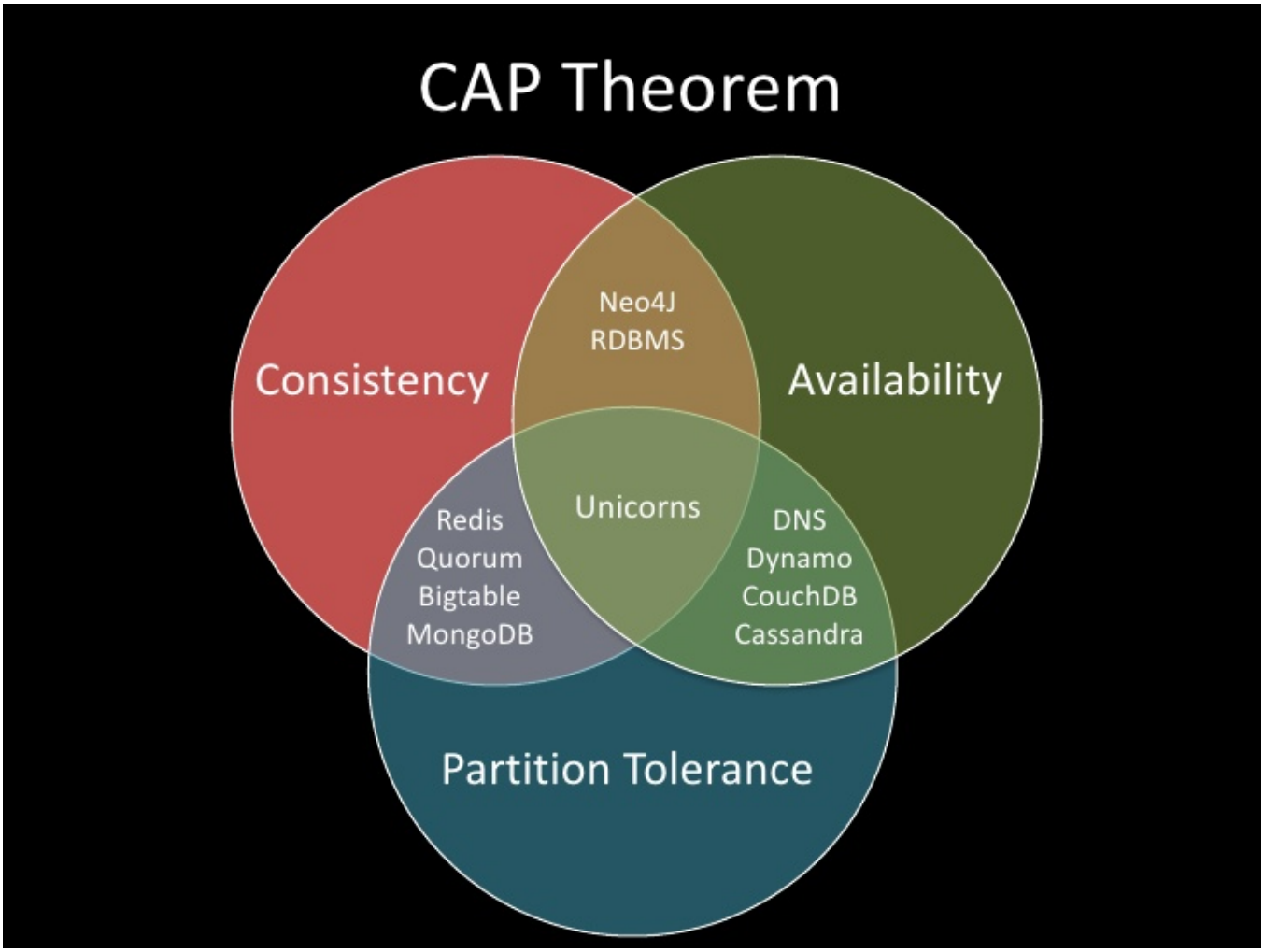

Figure 1.7 – CAP Theorem is a concept that a distributed storage system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.

acronym stands for

- **Basic Availability**: The system will appear to work most of the time, and there will be a response to any request. The response could still be 'failure' to obtain the requested data, or the data may be in an inconsistent or changing state.
- **Soft-state**: Stores do not have to be write-consistent, nor do different replicas have to be mutually consistent all the time. Data consistency is the developer's problem, and the data-store should not enforce it.
- **Eventual consistent**: It defines that if no update takes a very long time, all replicas eventually become consistent.

The BASE design encourages crafting operations in such a way that in the face of a network partition only a minor percent of the user gets affected. Independent and self-consistent operations can still make forward progress. There is no magic involved, but this does lead to higher

perceived availability of the system. In other cases where users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as a disconnected operation or offline mode, is becoming increasingly important. Some HTML5 features-in particular, on-client persistent storage-make disconnected operation easier going forward. The designer forfeits A in a way that users do not see. The users know that the client has scheduled the changes and will apply them to the data-store at a later time.

Chapter 2

Application-tailored storage

Analysts are expecting that by 2025 the global datasphere will grow to 163 zettabytes, an increase of more than 1000% from the 16.1 ZB of data of 2016. That puts pressure not only to hardware manufacturers for device capacity and performance but also to data-stores for handling this Big Data growth. The demand for performance, scalability, and workload diversity of modern applications [7,28,86,129] has led to the development of a broad spectrum of storage platforms. Nowadays the storage inventory is highly diversified with parallel filesystems for concurrent access [19,94], scalable object storage for capacity and scalability [5,23,130] and databases optimized for use in memory [26,35], hard-disks [36,39], or non-volatile memories [46,57,73–75].

These data-centric platforms assume that data is the primary and permanent asset, and applications come and go. In the data-centric architecture, the data model precedes the implementation of any given application and will be around and valid long after it is gone. Many people may think this is what happens now or what should happen. However, that is very rarely the case. Businesses want functionality, and they purchase or build application systems. Scientists wish to simulate physical phenomena, which may be computationally intensive or I/O intense. Each application system has its data model and inextricably tied code with it. Thereby, it is challenging to change the data model of an implemented application system, as there may be millions of lines of code dependent on the existing model. Such claims led the database community to step away from relational databases to NoSQL databases. A NoSQL database lets a developer build an application without having to define the schema first unlike relational databases which require to specify the schema before adding any data to the system. No predefined schema makes NoSQL databases more natural to update as the data and requirements change.

The first golden standard on applications is portability; the property of changing the system that stores the data without changes in the application source code. The characterization "general purpose" can be quite misleading as it refers to the system being agnostic to the application built upon it, as well as to the type of workloads the system can serve.

Many of the problems a data-store tries to solve are antagonistic. For example, strict ordering involves high synchronization overheads that compromise concurrency and scalability. Large files favor sequential access with high throughput whereas small files favor random seeking with

low latency. The conventions and the model upon which a system relies impose certain limitations on the type of workload the system performs optimally for; no system can perform equally for every workload. Therefore, it is the duty of the developer, or application administrator, to choose the best-fit system for the application. Factors such as provided abstractions (e.g., structured or unstructured data), functionality (e.g., consistency model, capabilities), and application workload (e.g., transfer size, random or sequential I/O), can affect that decision. That decision only reflects the state for a particular point in time, without guarantees that the "best-fit" match will hold over a long period. For example, workloads may change as the application evolves, the system performance may degrade over time, or cost-related policies may necessitate a vendor change. Portability is essential to avoid being locked-in to a particular vendor.

Even with portability ensured, the application is still susceptible to bottlenecks and outages inflicted by the use of a single system, regardless of the system. So, that leads us to the second golden standard, interoperability. It refers to the ability to leverage more than one platforms in parallel. Doing so unlocks complex operational strategies that would not be feasible otherwise: capability-aware workload distribution for cost reduction or quality of service reasons [103, 109, 133], application scalability independent to a particular platform, and circumvention of provider policies such as size quotas or local governmental laws regarding data content or encryption. Also, dispersing data to multiple locations shields the system against offline attacks, since no single vendor contains the full dataset. The cost for such strategies is the requirement of an adequate mechanism to orchestrate the multi-platform data distribution.

A potential way to implement this mechanism is to use existing overlay data-stores such as Ceph [130] and Glusterfs [100], on a per-application basis. Up to a point, such functionality is natively provided by existing data-stores as dedicated namespace and storage pools. As happens with any of the existing systems, when the application invokes an I/O request to the system, the developer completely loses control of the data. How the system will respond to the request is hardcoded into the source code and cannot change without deep and meticulous intervention. An alternative would be to implement all the necessary I/O mechanisms within the application. Such an arrangement must at least perform parallel I/O across diverse platforms, error handling, data reconstruction, and support concurrent I/O requests arriving from the application. Building this mechanism is a rather complicated task that requires expertise on I/O handling and parallel programming. Inevitably that makes applications much more cumbersome and, more importantly, every new application must reimplement the same mechanism. This dissertation introduces “application-tailored storage systems’, “storage containers’, as a distributed storage middleware with programmable behavior. The middleware separates I/O logic from the rest of the application logic, thereby separating changes made to application codes by science users from changes made to I/O actions by developers or administrators. Such design helps to defer I/O decisions until the deployment phase, which is the key for portability, workload tuning, and data post-processing transparently to the application. I/O logic design is external to the application, thus allows cleaner code while the developers remain in control. For the middleware development, we introduce *Tromos SDK* (Software Development Kit): a set of highly compartmentalized

modules that provide the building blocks for building customized distributed storage systems. Using its domain-specific language, developers can model and deploy systems with customized distribution logic, consistency, selection algorithms, data layout. Hence, building customized storage environments for the application is only a matter of choosing the appropriate schema. Figure 2.1 illustrate the concept of Storage containers.

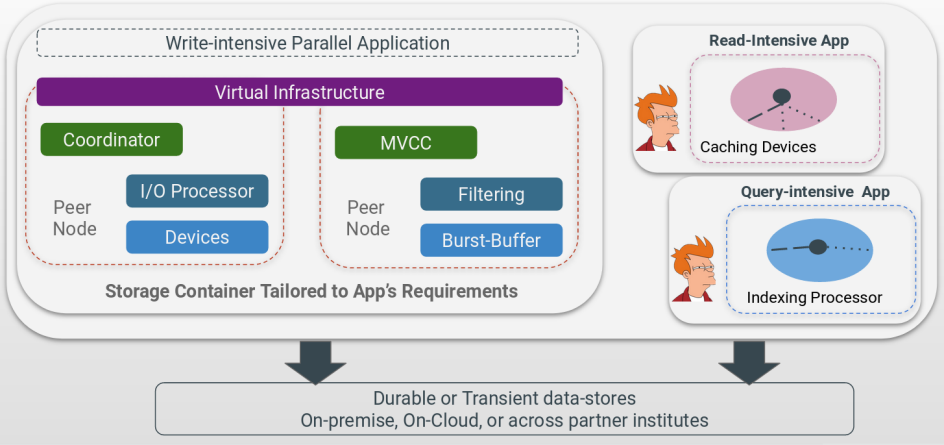

Figure 2.1 – Storage containers are middleware environments with data management tailored to the requirements of the application at hand. From the application architect's perspective, storage containers make possible to separate the I/O management from the rest of the application. From the perspective of the infrastructure's administrator, storage containers provide the mean to integrate I/O optimizations and improve resource utilization, without the need to intervene into the application's source code.

2.1 Application-tailored storage system

In this dissertation, we introduce Application-tailored storage (ATS) as client-side middleware for persistent storage that decouples application I/O logic from functionality logic. It allows developers to apply their I/O related policies without intervention in the source code. Being distributed, it can cluster clients to form a virtual storage infrastructure for a group of related applications. It is not meant to replace existing general purpose data-stores, but provide a data management layer that insulates application and allows it to scale independently. In the conventional notion, storage is the asset where applications come and go. Thereby, it is justified to apply one-size-fit-them-all policies to all the applications. ATS leans toward "application is the asset and can use any combination of existing storage systems". When the application terminates, so does the ATS. It allows developers to customize namespace management, consistency level, workload distribution, crash resiliency, and many others, on a per-application basis. Bellow, we present a list of use cases where ATS can be a valuable asset.

2.1.1 Portability

Claiming application portability across platforms entails predictable behavior across all the platform choices. Two distinct aspects often conflated are the API and the protocol. The API is a shared boundary across which two or more separate components exchange information. The protocol is a contract between components that define what is and is not guaranteed to happen on an API call. It defines the rules, syntax, semantics, synchronization of communication and possible error recovery methods. It does not define though how to realize them, i.e., implementation details. Along these lines, API is about portability on the source-code level whereas protocol is about portability on the binary-level.

Due to high synchronization overheads, filesystems in High-Performance Computing (HPC) tend to replace the strong Portable Operating System Interface (POSIX) semantics, with more relaxed semantics that favor scalability and concurrency. They are still accessible though through the same POSIX API. Thereby, two filesystems that expose the same API do not necessarily implement the same semantics [51].

The same holds for multi-cloud libraries [58, 65, 117]. They unify Cloud Storage provides on the API level, but not on the semantics. For API, unification means integration of the lowest common denominator of all vendor-specific API, i.e., the calls that are supported by all platforms. For protocol, unification means integration to the stricter of all vendor-specific protocols. From an application perspective, providing semantics stronger than those requested impose unnecessary synchronization overheads. Otherwise, semantics weaker than the requested can jeopardize application correctness [50, 51, 96, 110, 119, 126]. Evidently, the most appropriate "reference" protocol for an application depends on the application itself, and therefore general purpose systems, or libraries, cannot provide it, if they want to remain generic.

2.1.2 Federated Storage Toolkit

Architecturally, ATS is a federated collection of independent resources governed by a shared management system that handles data processing, distribution, and persistence [30,79]. Loosely coupled resources that act unilaterally, while being centrally managed, enable networks of virtually limitless capacities, eliminate disruptive outages, and circumvent bottlenecks and quotas.

Data-store for temporal data

Many HPC batch jobs do not require the strong consistency guarantees enforces by POSIX. Developers in that domain are turning to non-POSIX IO solutions because their applications are well-understood (e.g., distinct read/write phases, synchronization only needed during certain phases) and because these applications wreak havoc on file systems designed for general-purpose workloads (e.g., checkpoint-restart). For example, BatchFS [136] and DeltaFS [137], two filesystems optimal for batch-jobs, perform more client-side processing and merge updates when the job finishes. In computer architecture, Scratchpad memory (SPM) is a high-speed

internal memory used for temporary storage of calculations, data, and other work in progress. It is similar to the L1 cache in that it is the next closest memory to the ALU after the processor registers, with explicit instructions to move data to and from main memory.

Bringing the above together, ATS makes a convenient tool for creating Scratchpad persistent storage using the local storage of compute nodes. Doing so, nodes that run batch jobs can use this temporary storage instead of having to communicate with the parallel filesystem (e.g., Lustre, GPFS). Using ATS, nodes that need to store temporal data can also use the storage capacity of their neighbors, instead of being limited to the host boundaries. To this end, ATS can also help to improve performance. For example, when a node completes a calculation, it can store data to the node that is most likely to request them shortly. Thereby, it takes only one transfer over the network compared to storing to parallel filesystems, which would take one write and one read.

Tiered Storage Management

What will likely characterize Exascale -computing systems capable of a quintillion calculations per second- is three to four orders of magnitude increase in concurrency from current standards, a substantially larger storage capacity, and a deepening of the storage hierarchy [56]. At the same time, technology advents in memory, storage devices, and network introduce new challenges and opportunities in tiered storage management.

The current practice of independent optimization on each layer of the system I/O software stack will not scale to the new levels of concurrency, storage hierarchy, and capacity. ATS offset a comprehensive solution for managing multiple storage tiers in a distributed setting. That contradicts past works which were handling storage tiers in pairs, with the upper tier acting as a staging area for the lower tier. *Tromos SDK* exposes static (e.g., tier, device type, capabilities, location) and runtime (e.g., load, access pattern) information onto higher layers for making intelligent data placement decisions. The merits of heterogeneous tier management have been previously demonstrated both for performance in cluster deployment and capacity provision on single-host deployment.

OctopusFS [61] automates data management across nodes and tiers in order to improve throughput and cluster utilization. It includes a variety of pluggable policies for automating data placement, retrieval, and caching across the storage tiers and cluster nodes. The policies employ multi-objective optimization techniques for making intelligent data management decisions based on the requirements of fault tolerance, data and load balancing, and throughput maximization.

HetFS [63] demonstrates tiered storage management on a single host. It is an extension to ZFS filesystem which provides an intelligent mechanism that balances the benefits and drawbacks of each tier according to preprogrammed filters. For HetFS, a tier is a physical medium with distinct characteristics. For example, hard disk drives (HDDs) or magnetic tapes have large capacity, high sequential throughput, but induce high latency; solid-state drives (SSDs) favor the degree of parallelism and random operations, but have limited write cycles; and non-volatile memories (NVMs) outperform NAND-based SSDs but are expensive the introduce a full new set

of idiosyncrasies.

2.1.3 Second-order management

A commonly conflated term is “scalability”. Vendors colloquially use it to refer to the expansion ability of their infrastructure, not to the resources themselves, i.e., provide more resources, not faster. Thereby, if one can realize an application out of these building blocks, then all it takes to make it scalable is keeping functionality within the performance boundaries of the resource. Boundaries can exist in various aspects, such as throttling on data transfer (i.e., requests per second) or quotas (i.e., fixed capacity). Applications can either comply with those limits and scale no further or do their custom second-order scaling and scale beyond the boundaries of a single resource.

For example, cloud bursting is an application deployment model in which an application runs on a fixed pool of on-premise resources and, during peak-load, it spans onto additional cloud resources. The advantage of such a hybrid cloud deployment is that an organization only pays for extra compute resources when they are needed. It is a way to optimize resource usage (i.e., save on costs) and still provide access to capacity when needed.

ATS heavily relies on a microservice architectural pattern, components that are far less state dependent upon one another than monolithic applications of the past. That makes it feasible not only to add more resources and scale up the application but use resources from different vendors. With criteria governed by functionality, cost-effectiveness, or flexibility demands, developers can choose the best cloud offering and seamlessly integrate it to the application.

Multi Cloud Usage

A common concern when organizations decide to go to the cloud is the risk associated with dependency on one external firm, such as Amazon, Google or Microsoft. Avoiding vendor lock-in while balancing the feature benefits of multiple platforms has been the name of the game from that perch for more than 30 years. In response, it makes sense to minimize the perceived risks by using more than one cloud provider. That provides an additional option in case the relationship with one provider becomes untenable for some reason, either that be financial, or technical such as service failures or outages. The multi-cloud strategy can be an enabler for mitigating vendor lock-in risks as mean to minimize their dependence on any one provider and ensure they are not locked into a single contract, lest they miss out on some new development from a competing Cloud provider. It allows applications to meet specific workload or application requirements by selectively consuming unique capability or services from several platforms, without any means of connection or orchestration between them. ATS standardizes as much as possible a single management and monitoring tool for cloud interoperability, offloading applications from having to deal with different management portals and processes.

Provider-independent security

When data cross the boundaries of a data-store, applications immediately lose control of it and delegate data integrity and security to the vendor. Bugs, misconfigurations, or operator error can accidentally expose, or even corrupt, data. Criminals routinely gain unauthorized access to corporate servers. Even more insidious is the fact that the employees themselves sometimes violate customer privacy out of carelessness, avarice, or mere curiosity.

Content-addressable storage (CAS) may prohibit modifications to content but still allows read access to the content, i.e., ensures security but not privacy. Encryption may discourage unauthorized access but cannot prohibit it. Data are still accessible if attackers pay effort on the decryption. The proposed solution to render data immune to off-line attacks is to encrypt, split, and disperse data over multiple providers; thus no single entity has the full content. That also improves performance since data are in-parallel retrievable. The only way to gain access to the content is to own the logical file descriptor, pointing to chunks and including the encryptions keys.

Moreover, one can encrypt chunks with different keys so that even on key leaking, only partial reconstruction is possible. That, however, induces certain availability constraints - if any of the providers go down, full data become unavailable. The solution is to apply erasure-coding after encryption; an encoding method which transforms data into K+P streams, out of which only K (any K) suffice to reconstruct the original data. Tahoe-lafs [131] has been doing for almost a decade for filesystems, and data-stores like StorJ, Sia, Maidsafe, and Filecoin have been trying lately to combine Blockchain and Cloud storage potentials.

2.2 SDK for distributed storage systems

The challenge of constructing application-tailored storage middleware is that every application must build its custom environment, meaning that every application must "reinvent the wheel" with subsequent costs in time, money, and resources. To address that developmental challenge we introduce *Tromos* Software Development Kit (SDK); a collection of I/O process components and meshing languages, used as building blocks for the construction of distributed middleware. It better identifies as a community-driven suite where team members can push, fetch, edit, review, and version, I/O related components [49]. *Tromos* advocates separation of concerns, separation of the data path from the control path, and modularity. The principle idea behind *Tromos* is to realize distributed middleware as a composition of basic components, without systems programming. For this reason, the composition is distinguished in three viewpoints:

- **Component view**: Built-in plugins for a wide range of data processing and request handling. The components are narrow-scoped, down to the level of the algorithm. The focus on this view is to provide a versatile bench so to support various algorithms implementations. For example, provide cross-layer information to device selection component so to make intelligent decisions, or provide the necessary context for the component to store and load its

state.

- **Driver view**: Focuses on how to use components as building blocks for higher-level elements. We achieve it by using declarative language that represents those elements as stacks or graphs of pipelined components.
- **Service View** Synthesis frameworks for composing the abstract Driver definitions into running instances. That provides far greater flexibility for abstraction than API abstraction as it allows easy extensibility and customization without the need to develop or change the underlying implementation code.

Tromos architectural pattern models and directly manages the data, logic, rules, and structure of the system, independently of the user-visible interface. The goal is to enable developers and administrators to create customized storage environments optimized to the underlying infrastructure and the application requirements, without changes in the application.

To address that, *Tromos* provides an infrastructure configuration language, called *Manifest*, used to design, document, and describe the target storage environment as a code. Configuration files created with *Manifest* constructively specify distributed storage middleware as compositions of building blocks, as Docker [81] does for containers. Such codified documentation simplifies many aspects of storage provision, configuration, analysis, and reasoning about performance and correctness issues that otherwise would demand deep intervention and scavenge on multiple subsystems. Centralizing the model in the *Manifest* makes it easier to reason about the system behavior and find "bugs" in the same way one can reason about an application behavior by looking into its source code. In particular, it promotes a high-level, functional style of programming that improves communication with domain experts, which is one of the hardest problems in software development and research.

Tromos Deployer treats *Manifest* files as executable code for which it generates an execution plan describing what it will do to reach the described target environment. Administrators can preview and validate the changes before they are applied, in order to mitigate undesired side-effects. The validation phase draws clean lines between deployment and testing. Errors in automation scripts will have just the same impact as errors made during manual deploys. Every time the *Deployer* applies a model, it generates and configures the same **idempotent and reproducible target environment** without user involvement [48, 99].

In-vitro experiments Inherent composability and reproducibility properties of systems designed in *Tromos SDK* are desired not only in business but in academia as well [59]. A common practice for researchers that want to validate a new algorithm is to integrate the new algorithm into existing platforms and measure the outcome over the unmodified version. Although theoretically correct, this approach is impractical. On the one hand, researchers must spend a significant amount of time browsing through the source code to find where to place the modifications. On

the other hand, intervention in a hardly coupled pipeline can cause side-effects (e.g., race conditions) making it difficult to reason whether performance fluctuations are due to the new algorithm or side-effects. *Tromos* modular design enables researchers to integrate their algorithms as plugins without requiring in-depth knowledge of the internal system functionality. The same also helps to evaluate algorithm performance, objectively and free of side-effects. To this end, potential *Tromos* usage extends from strict software-defined storage to an in-vitro emulator for stressing components, both in isolation and in a pipeline, to guarantee a predictable behavior when deployed.

2.2.1 Components

Conventional storage systems are "monolithic" black boxes that do not share any code, despite sharing similar functionality. In *Tromos* we identify and compartmentalize into modules the most common tasks found in the majority of storage systems. Among others, we identify components for external data-stores and databases (e.g., filesystems, object storage, key-value), resource selection, data processing, data distribution, distributed synchronization and consistency, data reconstruction, and data layout. *Modules* aim to be loosely coupled, virtually cause no side-effects, and have minimal inter-module dependencies.

Tromos advocates small and reusable modules that are easy to test, validate, and reason about their behavior, both in isolation and in interaction with others. That is without saying that module integration can happen unconditionally. Based on our experience it is imperative to validate modules in various topologies (meshes) as certain behaviors appear only under specific conditions (e.g., when a stack includes more than two modules). *Tromos* follows programming conventions that make modules available both as libraries and as plugins. Plugins make it possible to create custom pipelines for I/O processing dynamically. It also exploits Golang [29] capabilities to provide dry runs for unit testing and profiling.

Services The *Services* are synthesis frameworks that jointly provide primitives for building distributed storage systems. The objects of those *Services* are independent, and potentially geographically distributed processing units. *Device* services are about portability and storage provision, *Datapath* about parallelization and in-transit processing, and *Coordinators* about namespace management and distributed synchronization. The principle idea behind *Services* is to provide generic primitives whose semantics are not hardcoded, but governed by auto-produced drivers. Provided domain specific languages define *Drivers* as compositions of pipelined components. By controlling the pipeline structure, or mesh, developers can describe customized drivers and adjust *Service* behavior according to the application needs.

Devices *Devices* are about storage provision and portability across a broad set of underlying data-stores. They aim to bring the benefits of Active Storage to cloud-world. *Devices* could, for instance, run compression and aggregation functions, throttling, deduplication, and optimize

data layout to minimize the bandwidth consumption. *Resource* language defines a *Device* as an asynchronous stack of processing modules categorized into Connectors, Translators, and Proxies. *Connectors* unify data-stores on the API level; *Translators* are processing layers that unify data-stores on protocol level and optimize I/O concerning data-store capabilities; the latter makes a *Device* accessible to clients through the network.

I/O Processors *Processors* are about in-transit I/O processing of application output. They aim to decouple the application business or computation logic from the I/O logic. *TrIO* (Transparent Regulated I/O) language defines a *Processor* as a graph of processing modules. Automated execution of the model then leads to less low-level concerns and burden for the developers. It automates, for example, module bridging, parallel execution, synchronization, and many others that require knowledge in parallel and distributed programming. Modules may apply inline transformations to the stream (e.g., compression, filtering, indexing) or create complex structures with branches and multiplexers (e.g., strip, mirror). That allows a single client to transfer data to multiple *Devices* concurrently. *Processors* can form a composite processing network for moving filtering logic from computation nodes to I/O nodes, closer to the data.

Coordinator Service *Coordinators* are about namespace management, metadata management, and distributed synchronization. Using the *Coordinators*, concurrently running clients can access the same data without jeopardizing correctness. *Keyzone* language defines a *Coordinator* as a synchronous stack of processing modules. From an abstract point of view, the language is very similar to the *Resource* language used in *Devices*. It also categorizes modules into Connectors, Translators, and Proxies. The *Connectors* implement ledger functionality on top of external databases. A ledger is a shared log that durably persists intentions and updates of top-level operations. Consistency, or an agreed global ordering, becomes simply the order of events in the log. *Translators* controls how requests access the ledger. By controlling the *Translators*, developers can enforce the adjust *Coordinator* to the correctness requirements of the application. For example, a sequencer in the stack will block incoming requests as long as the previous request is active, while on its absence the requests will be running in parallel.

2.2.2 Virtual Storage Infrastructure

Middleware built upon *Services* can scale by adding more instances of the appropriate *Service* type. *Tromos* uses a thin layer acting as persistent storage interfaces, to separate user-visible interface from low-level data-management, written in *Service* API, i.e., to separate I/O handling logic from the application computation or business logic. Storage aspects such as request concurrency, sessions, in-transit processing, placement decisions, data distribution, data layout, and reconstruction, are externally programmable in a language called *Manifest*. With that, developers can tailor the middleware according to the application needs by describing the desired target environment as a composition of custom data-management routines. The *Manifest* identifies

three grids; the messaging, the data, and the processing grid. Its grid run independently and in principle include all the necessary information for the client-side middleware on how to handle the available services.

Chapter 3

Programmable Devices

In this Chapter, we introduce a framework for building virtual object storage *Devices* atop diverse storage platforms. The framework distinguishes the data plane from the control plane. Using the provided declarative language the developers can model the device properties as a stack of basic components. The framework automates the synthesis and produces a running instance of the target *Device*.

3.1 Objectives

In order to transfer data to and from a data store, the applications must incorporate into their source code the API calls. That means the application must also know the constraints, semantics, synchronization of communication, and the request/reply syntax. That becomes even more complicated as the application must also implement the data-management logic. In our design, the *Devices* become the intermediary for running out-of-band data-management functions. Using composable drivers, the developers can customize the *Device* semantics according to the application requirements, without the direct input or any management from the application. Existing drivers paralellize access, merge requests, and throttle requests, before forwarding data to external data-stores. Next, we present a few of the challenges *Devices* are trying to address.

3.1.1 Portability

POSIX standard dominated filesystems for many decades. It defined a programming model to ensure application portability across different implementations of POSIX-compliant filesystems. However, there is no such notion in the newly born and emerging Cloud Storage world. Vendors like Amazon S3, Microsoft Azure, Google Cloud Storage are all trying to promote their solutions and APIs. Subsequently, if applications incorporate vendor-specific API, they bound to that vendor. To avoid vendor-lock, Multi-cloud connectors such as libcloud [65] for Python, jclouds [58] for Java, and Stow [117] for Golang provide a unified API with the "lowest common denominator" [95, 112] of features available in all platforms. Under the hood, connectors translate the

unified API calls to the vendor-specific calls. The drawback is they hide vendor-specific capabilities from applications. Examples include asynchronous operations, notifications, configuration management, monitoring, and policies. The pitfall is that even a unified API is not a guarantee that the underlying storage system will exhibit identical behaviors. Many of the problems a data-store try to solve are antagonistic. For example, strict ordering involves high synchronization overheads that compromise concurrency and scalability. Large files favor sequential access with high throughput whereas small files favor random seek with low latency. A potential solution would be for applications to integrate codepaths for every support vendor but that would complicate the development and maintenance effort significantly.

Device framework follow a holistic approach taking the whole storage stack into account and enable developers to drive the integration effort at a "higher denominator" driver. Such drivers retain access to all the robust features and rich sets of services the cloud offers and at the same time provide an integrated mean for applications to access diverge storage platforms, i.g., achieve portability.

3.1.2 Differentiated Content access

Many in critical areas of science and technology are becoming more and more data intensive. These applications transfer large amounts of data from storage nodes to compute nodes for processing, which is costly and bandwidth consuming. The data movement often dominates the applications' runtime. *Device* framework holds a promise for high performance I/O for these applications by moving appropriate I/O logic from compute nodes to storage nodes. *Devices* allow to run arbitrary code on them and provide differentiated content access over a shared platform. Overly simplified, *Devices* is a virtualization layer isolate and provide distinct characteristics to the collection on the platform (e.g., directories, buckets, or set of keys). Let us draw an example with a shared platform with directories A and B. A write-intensive application will use device A to aggregate requests before flushing them to directory A. In contrast, a read-intensive application will use device B with prefetch and caching capabilities to alleviate platform from continuous read requests.

Device framework can map collections on shared platform onto higher-level virtual object devices in the same manner device mapper [101] maps physical block devices onto higher-level virtual block devices. Its kernel-based nature though makes it inadequate for use in cloud-storage and high-throughput physical devices such as non-volatile memories. In contrast, *Device* framework does not suffer from these deficits as it resides entirely in user-space. It can further leverage zero-copy technologies to avoid double copies and transitions from userspace to kernel space, that hinders performance seeking applications.

3.2 Device Driver Synthesis

A *Device* driver is a stack of pipelined handlers. They are fundamental components that apply policies to incoming requests and pass them down to the next layer the stack. They are agnostic to the request payload and do not apply any transformation on it — policies at this level care about how to write data to the lower layers, not about the content. *Handlers* can forward, divide, or merge incoming I/O requests before calling the lower handler.

For a *Device* we take for granted three things

i will be hit by concurrent requests that belong on different entities (e.g., different file handlers)

ii will be hit by concurrent requests that belong on the same higher level entity (e.g., a file handler)

iii there will be multiple handlers in the Stack.

Starting parallel operations on non-thread safe drivers can result to race conditions. One solution to guarantee correctness is to deal with complex (and usually inefficient) locking schemes. The other solution is to create a new driver instance inside each thread. Based on that, we built the *Device* API around organizational constructs termed *channels* and *streams*. From the client perspective, a channel is a dedicated driver instance to which it transfers data in streams. From the *Device* perspective, a channel is as a sequence of operations (handlers) that execute in issue-order and terminate to a private collection on the backend (e.g., a directory for filesystems, a bucket for object storage, or a prefix for key-value databases). A stream is a variable-length sequence of bytes that terminate on a data holder within the collection (e.g., files, objects, keys).

Code: 3.1 presents a skeleton of a *Device* handler. In order for a handler to participate in the stack it must include a public “device.Stack” field. The framework used that field to place the handler as a layer in the stack. The NewChannel method is about creating an isolated channel within the handler. We explain more details about channel functionality in the next Section.

3.2.1 Modeling Language

The *Resource* is a YAML-compatible language for composing drivers for *Devices*. It consists of a unique *Device* identifier, functional information such credentials and paths, and a stack of asynchronous handlers (Code 3.2). Handlers, or stack layers, are called by parents and in turn call children. Some layers are “final” in the sense that terminates requests instead of passing them on, so they do not have children. For example, /device/proxy/client forwards requests to another node through the network, /device/connector/POSIX invokes filesystem operations. Others are “initial” and inject requests into the system from elsewhere, so they do not have parents. For example, device/proxy/server injects requests from the network. Intermediate layers have one parent and one child.

```
type Layer struct {
    // Used by synthesis service to pipeline the handler
    device.Stack
    // Channel tracking
    channels    []device.Channel
    // Some internal device information
    dinfo  struct{}
}

func (l *Layer) NewChannel(name string) (device.Channel) {
    // Create a cross-layer channel
    c := &Channel{
        // Link local with remote channel
        callback: l.Stack.NewChannel(),
        // Initialize a metadata lookup on the channel
        future: []device.Stream{}{},
    }
    // Return channel to caller, but keep track of it
    l.channels = append(l.channels, c)
    return c
}
```

Listing 3.1 – Device Handler skeleton. All handlers must export the device.Stack variable. The framework use this field place the handler into the pipeline. NewChannel() is one of the methods a handler must implement in order to be stackable and compliant with the Device API. It must be noted that channels are created synchronously, layer by layer

Figure 3.1 depicts the synthesis process. According to their position in the stack, we differentiate the handlers to the following categories

Connectors

Connectors are the lowermost handlers in the stack. Their purpose is to communicate with different provider APIs over transport protocols including system calls, library calls, REST API calls, or other network protocols (e.g., iscsi). In order to track and verify the upload progress, they annotate transferred blocks with an MD5 hash. If a transfer has failed, they can resend blocks are required. Available connectors include:

- **Filesystem** a syscall-based connector for accessing locally mounted filesystems.
- **Consul** a REST-based connector for accessing Consul [47] distributed key/value store.
- **Googledrive** a REST-based driver for accessing Googledrive object storage. To cope with the eventual consistency, we use polling at regular intervals to guarantee the success of data transfer. We acknowledge the request only when data become visible. Due to the policing inefficiencies, we plan to reimplement the connector using provided notifications.

- **Stow** a multi-cloud connector written in Go [117] that allows access to Amazon S3 and Microsoft Azure Blob Storage.

Translators

Albeit most of the work performed in the Connectors is not CPU intensive, it is I/O bound. To compensate the waiting it is desirable to perform more operations in parallel (e.g., multi-part upload for bandwidth utilization), improve I/O efficiency (e.g., data aggregation for throughput), and minimize communication (e.g., data rearrangement for fragmentation). Problems like this are usually solved on a higher level using threads or async libraries such as Twisted [34], Tornado or gevent.

In our approach, we address the above within the scope of the *Device* by using *Translators*. They are processing layers located just above the *Connectors*, whose purpose range from providing thread-safe access to the connector, until making the driver compatible with the platform limitations [4, 42]. In general, the *Connectors* are meant to unify platforms on the API-level and the *Translators* are meant to unify platforms on the protocol level. Available connectors include:

- **Throttler** regulates data traffic using the token-bucket method [60, 106]. Platforms that impose limitations on the number of incoming requests per second will start rejecting requests after a certain limit. We use throttlers to comply with these limitations and avoid erratic behavior, as perceived by clients.
- **Chunker** disaggregates incoming request to multiple outgoing requests. We use chunker to circumvent constraints as to the maximum supported file size on the platform. More specifically, it breaks input data to multiple chunks sized up to the maximum supported size. Another usage is to upload multiple chunks in parallel so to decrease upload time, even for platforms that do not natively support such functionality.
- **Burst buffer** aggregates incoming requests into a single outgoing request. We use a burst buffer to and temporarily store the data of requests in order to insulate the persistent storage from bursty workloads [14], characterized by the massive amount of small I/O operations. Another usage of this operation is to improve bandwidth utilization when transferring data to a Cloud provider, by batching multiple requests into a single message.
- **Prefetcher** speculatively fetches and caches data. Applications that exhibit spatial locality are likely to ask for adjacent blocks of data shortly. We use prefetcher to bring data in advance into the host memory, so that future Read() does not have to sustain queueing and transfer overhead.

Proxy *Proxies* render *Devices* accessible through the network. Clients only use *Proxies* to negotiate an out-of-band connection for the actual data transfer. That is to avoid encapsulation, fragmentation, and other overheads associated with structured data transfer. Currently, the

available Proxy relies on Remote Procedure Call (RPC); a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. We base the implementation on an open-source universal RPC framework called gRPC [41] and flatbuffers [40] for serialization. In Linux, port 0 is used to let the kernel dynamically assign a new port to the request, thus enabling greater flexibility than hardcoding the ports. For the data transfer available transport protocols: TCP, UDT, and KCP. Complying with IANA guidelines, if more than one *Devices* exist per host, their proxies must start port 49152 onwards. For example: if there are five *Devices*, the respective ports will be from 49152 to 49156 open. These ports are where the RPC server is listening for requests, not the ports that data transfer will take place.

Capabilities *Capabilities* are key-value pairs used as labels for the supported functions of the driver. They can annotate physical properties of the underlying platform (e.g., HDD-based, SSD-based), hints for the optimal workloads of the platform (e.g., small writes for key-value databases, sequential for filesystems), geographical or cluster-wise location of the node running the *Device*, or any other functional hint. *Tromos* use these labels in the *Device* selection process in order to provide tiered storage management with quality of service guarantees. For example, placement policies may aim at sequential throughput and favor *Devices* atop HDDs or random access and favor *Devices* atop SSDs.

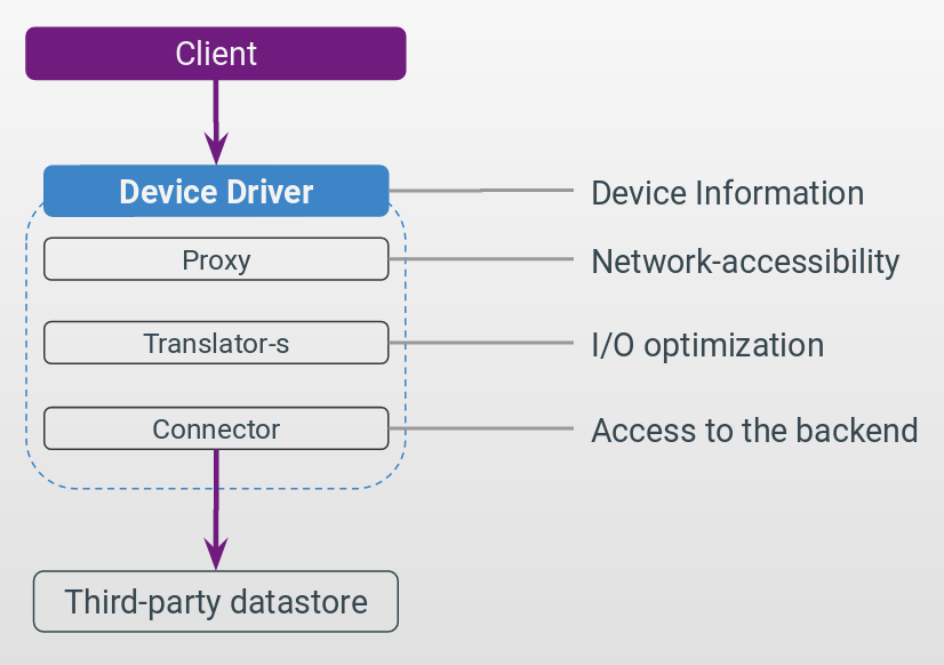

Figure 3.1 – Synthesis Service for Device drivers. It piggybacks components from the repository into an asynchronous stack. The Connectors unify underlying the data-stores on the API-level. Translators handle data-request and unify data-stores on the level of semantics. Proxies make the *Device* available through the network.

```
# Device identifier
"dr3":
    # Device Characteristics (used for Device selection)
    Capabilities: [ "default", "scannable", "HDD" ]
    # Data-store client
    Connector:
        plugin: "filesystem"
        path: "/scratch/vol33"
    # Request-processing layers
    Translators:
        # Numberical-ordered stack
        "0":
            plugin: "throttler"
            rate: 120M
            capacity: 1B
            regulate: channel
        "1":
            plugin: "burstbuffer"
            blocksize: 1M
    # Make device accessible through the network
    Proxy:
      plugin: "grpc"
      service: "10.200.0.5:7773"
```

Listing 3.2 – Composition of customized Device driver in the Resource language. The exposed capabilities are used as hints in the resource selection process. The translators are stacked according to their numerical order.

3.3 Device Runtime

Devices adopt a two-phase I/O strategy [70, 80] that separate the data transfer phase, between a client and the *Device*, from the I/O phase between the *Device* and the backend storage. The runtime entities and the interaction are depicted in Figure 3.2.

3.3.1 Channels

Before start transferring data to and from the *Device*, the clients must establish an end-to-end channel with it. The channel creation is synchronous. Every layer that receives the creation request initializes a local channel and forwards the request down the stack. The process continues until the request reaches the *Connector*. When this happens, the *Connectors* creates a new collection on the backend and replies the request with the collection identifier. Any potential conflicts with existing collections occur at this step, thus removing time-consuming checks from the critical I/O path. The reply propagates the stack upwards, with every layer linking its local channel to the channel of its lower. Once the reply reaches the client, a cross-layer channel is established, identified by the collection identifier.

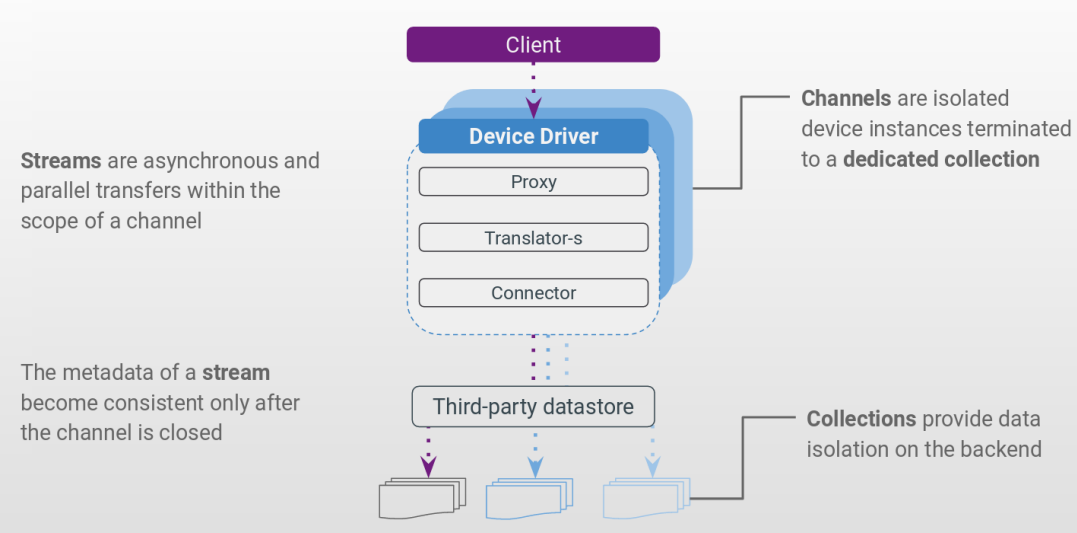

Figure 3.2 – Outline of the Device runtime

Channels are bind with cancellation contexts so to terminate at any time. *The context is a structure which carries deadlines, cancelation signals and other request-scoped values across API boundaries and between processes* [29]. The contexts are triggered explicitly (e.g., abort a handler-level operation) or implicitly (e.g., timeout). When this happens, the channel gracefully terminates its streams and removes the written object from the backend.

Grouping

In case of crash or abort, we must remove the partially written to reclaim the storage space. Crash handling mechanisms such as Journalling or distribution transactions maintain a persisted log of intentions and affected location so to recover the crash. Tracking every single stream requires those logs to be continuously updated. For journalling that translates to multiple I/O to the disk, and for the distributed transactions multiple round-trips over the network. Exploiting the fact that a channel direct all of its streams into the same collection, it suffices to know the collection identifier.

Synchronization barrier

The convention is that error handling goes to the level of the channel, not to the level of the individual stream. As a result, the state of the streams can be in flux until the channel closes. For example, even if one stream is successful, the channel will remove the persisted data if another stream in channel fails to persist. That is known as Release consistency model.

A possible analogy is that of a revision control system. The channels are like branches, streams like commits to the local branch, and channel closing is like pushing the local branch to a remote server. It is only after the commit that other clients can access the data.

3.3.2 Streams

Transferring data in block operations (e.g., write(), read(), put(), get()) is a burden because Linux applies limitations on maximum payload of a single operation. For synchronous threads, the consumer must continuously loop until EOF. For asynchronous threads, it is possible for the data of another thread to interleave the loop and cause buffer corruption. Streaming pipes provide an encapsulation that crosses the various layers with zero-copy capabilities, i.e., avoid copying data from one layer to another, if not necessary. Streams gracefully close on End Of File (EOF). As stream we denote an explicit data transfer from a client to the *Device* (Figure 3.3). That is to differentiate them from self-induced streams created by the layers in the stack. We refer to the latter as "sub-streams". Albeit the state can be in flux as long as the channel remains open, the stream metadata must appear in the correct issue-order. Central to the device API is the concept of a deferred, or future. It is a holder for metadata whose value is still unknown because I/O phase to the platform has not been completed yet [67]. Futures can be passed around just like regular objects that will block when asked for their value. Every handler embeds a lookup table for matching the (incoming) parent to the (outgoing) children requests. When a caller submits a request, the callee immediately replies with a *future* that points to an index within a metadata lookup. The assigned index is incremental and represents the invocation order of the parent request. It can be regarded as a "token" for the caller to later retrieve the (sub)stream metadata from that lookup. Futures permit the *caller* to progress with its other tasks, without stopping to wait for the data transfer to complete. Figure 3.4 presents the skeleton of a request-diving handler and how it uses the futures.

```
type Channel struct {
    // Linked channel on the lower handler
    callback  device.WriteChannel
    // Metadata lookup for asynchronous streams
    future    []device.Stream
}

func (ch *Channel) NewStream(src *io.PipeReader) int {
    // Forward the incoming request to the next layer
    return ch.callback.NewStream(src),
}
```

Listing 3.3 – Request forwarding (1 parent, 1 children). The handler does not take any action, it simply forwards the parent request to the next layer. Every handler maintains on its local channel a lookup table for keeping track of the outgoing streams. For every new stream, the channel returns a token for it to the caller. Because the stream are asynchronous, their metadata are not consistent until the channel is closed. When it closes, the callers can use the token to retrieve the stream metadata. This is done synchronously, layer by layer. Also see Figures 3.4 and 3.5

```
func (ch *Channel) NewStream(src *io.PipeReader) int {
    pr0, pw0 := io.Pipe()
    pr1, pw1 := io.Pipe()

    // Use lookup size as incremental identifier
    token := len(ch.future)

    // Initialize remote streams
    // Keep writing edge, forward reading edge
    s0 := &device.Stream{
        Index: ch.callback.NewStream(pr0),
        Parent: token,
    }
    s1 := &device.Stream{
        Index: ch.callback.NewStream(pr1),
        Parent: token,
    }}

    //  Track the children streams
    ch.future = append(ch.future, s0, s1)

    // initialize a new thread for pushing data
    // to writing edges of the pipes
    go func(){ /* Some data transfer or process logic */}

    // Data transfer continues on the backend
    return token
    // On Metadata() merge children with parent metadata
}
```

Listing 3.4 – Snippet of dividing a stream into two sub-streams (1 parent, N children). For every parent stream the handler returns a token. When the channel closes, the caller can call Metadata() method and retrieve the stream metadata using the given token

```
func (ch *Channel) NewStream(src *io.PipeReader) int {
    // Not shown - Initialize local buffer and pipes
    // Store source data to a local buffer
    n, _ := io.ReadAtLeast(src, ch.buffer[ch.boffset:],
        len(ch.buffer[ch.boffset:]))

    // Use lookup size as incremental identifier
    token := len(ch.future)

    // Track the children streams
    ch.future = append(ch.future, device.Stream{
        ID: token,
        Parent: commonStream.Index(),
    }}

    return token
    // On channel closing, flush the buffer
    // On Metadata() populate temporary metadata
    // with the metadata of persisted blob
}
```

Listing 3.5 – Snippet of merging streams into a super-stream (N parents, 1 grandparent). For every parent stream the handler returns a token. For every parent stream the handler returns a token. The super-stream gets persisted when the channel closes. Then, the client call Metadata() method and retrieve the stream metadata using the given token

3.3.3 I/O Phase

Connectors are dummy. They start transferring data to the platform as soon as they receive a request from the upper layers. Because that can happen at any time during the lifetime of a channel, we use the *Channel* closing as a synchronization barrier. Channels close synchronously in a top-down manner with each layer flushing and closing the lower layer. The metadata of all the previous operations become available when the closing request reach the connector. Because of the cascade effect -layers that create sub-streams- there is no simple relation between the logical client operations and the chunks written in the backend. To illustrate it with an example, a caller submits a parent request to a *Chunker* handler which then splits it to multiple sub-requests. Albeit *Chunker* would typically return more than one replies, one for every sub-request, the caller expects only one reply -for the parent request. Thereby metadata must be reconstructed layer by layer, with each layer merging the children metadata with the parent metadata. For the merge, the caller uses its embedded parent-to-children lookup and the metadata table returned by the callee.

Items An *Item* structure represent the metadata of a stream operation; it contains all the necessary information to reconstruct the data of a stream. Its format is "collection::[]object::[]offset",

where the [] symbol denote a list. The collection represents the group to which the channel was writing its data. The objects represent information about contiguous chunks written on the persisted storage in a single Connector operation. In case that chunks are not separate elements on the backend, but part of a larger blob the offset shows the chunk location within that blob. In this case, we use the term “grandparent” stream to denote the aggregation of multiple parent streams into a single stream (Figure 3.5). The *SDK* provides helper functions to translate the flat structure of children metadata (e.g., a vector of children metadata) to *Items* -append children to the chain of the parent *Item*.

Conventionally, storage systems return unique identifiers for addressing written data. Later, clients can use that identifier to retrieve the written content. When this happens, the system uses the identifier to find the address location - either algorithmically or with a lookup. Architecturally, such translation mechanism is not part of the *Device* but of the metadata lookup service. *Device* only return *Items* serialized in JSON packets. Alternatively stated, the *Items* are logical files pointing to data spread across various Cloud storage providers. Figure 3.3 presents an overview of the process, from channel creation until item retrieval.

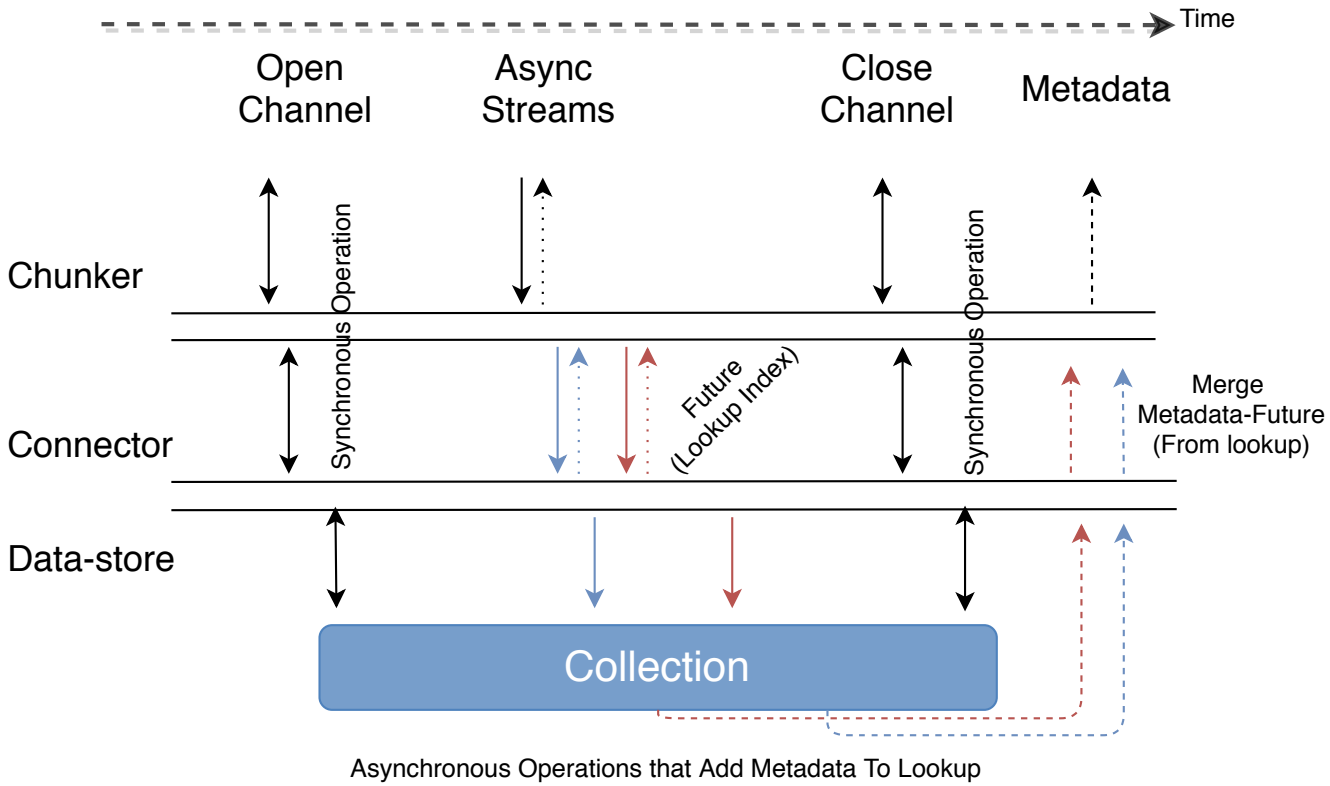

Figure 3.3 – Illustration of the writing process. First, the clients must open a channel to the *Device*. Within that channel, the client can start multiple asynchronous streams. For every stream, the channel returns a future(token) pointing to a lookup. When the channel closes the client can use the token to retrieve from the lookup the metadata of the stream

Device API	Return Value	Comment
Upstream(Name)	(Channel, error)	New upstream channel
Put(Channel, io.Reader)	(Future, error)	New write stream
Close(Channel)	(error)	Flush, close channel
Metadata([]Future)	map[Future]Item	Return items
Downstream(Name, Item)	(Channel, error)	Create channel
Get(Channel, io.Writer)	(Future, error)	New read stream
Scan()	([]string, []Item, error)	Index external data-store contents
Structure	**Fields**	
Channel	(Collection, Mode, Context)	Isolated view of the driver
Stream	(Index, Parent, Mode)	Asynchronous Data transfer
Item	(Off, Size, ID, Chain []Item)	Metadata for the data transfer

Table 3.1 – Stackable API implemented by the translators and the connector. Remove(), Info() and other calls are emitted for abbreviation.

3.4 Summary

In this Chapter we present a framework for composing *Drivers* for active storage *Devices*. The *Resource* is a declarative language that abstracts *Drivers* as a stack (pipeline) of basic handlers. That makes it easier to understand and reason about, less buggy, easier extendable compared to hardcoded drivers. Drivers can normalize data-stores, tune I/O handling to the expected workload, and harness performance benefits by moving functionality closer to the data.

The *Device* API is built around the concept of *Channels* and *Streams*. The *Channels* are cross-layer constructs that appear to clients as different and isolated instances of the *Driver*. Clients can use channels to avoid dealing with complex locking. The *Streams* are asynchronous channel operations for transferring sequences of bytes with arbitrary length. Every *Stream* is associated with an *Item*; a token (future) to represent the results of asynchronous *Streams*. When the channel closes, the *Item* is populated with all the necessary information to reconstruct the original data.

3.4.1 Related Work

The device mapper is a long-living Linux kernel framework that maps physical block devices onto higher-level virtual block devices. It forms the foundation of the logical volume manager (LVM), software RAIDs and dm-crypt disk encryption, and offers additional features such as file system snapshots. Linux storage devices are modular and stackable. That promotes flexibility and allows independent development efforts, but leads to a vast number of possible configurations. That requires the user to manage the stack because there is not enough commonality to enable effective automation. Over time, various tools emerged claiming easier setup and management without requiring expert-level storage administration knowledge. Such tools range from local management to cloud management.

In the past ten years, volume-managing filesystems (VMFs) such as ZFS and Btrfs have come into vogue and gained users, after being previously available only on other UNIX-based operating systems. These incorporate what would be handled by multiple tools under traditional Linux into a single tool. Redundancy, thin provisioning, volume management, and filesystems become features within a single comprehensive, consistent configuration system. Where a traditional Linux storage stack exposes the layers of block devices to the user to manage, VMFs hide everything in a pool.

Stratis [98] is a local storage solution that lets multiple logical filesystems share a pool of storage allocated from one or more block devices. Instead of an entirely in-kernel approach like ZFS or Btrfs, Stratis uses a hybrid user/kernel approach that builds upon existing block capabilities like device-mapper, existing filesystem capabilities like XFS, and a userspace daemon for monitoring and control. The goal of Stratis is to provide the conceptual simplicity of volume-managing filesystems and surpass them in areas such as monitoring and notification, automatic reconfiguration, and integration with higher-level storage management frameworks.

ARIA [112] is a tool developer can use to deploy and orchestrate a single application on multiple infrastructures. Is built on three pillars that are needed to manage the entire stack and lifecycle of an application while circumventing the need for a single abstraction layer. ARIA is the closest to our framework, but there is a subtle difference that makes them completely different projects. ARIA is lifecycle management tool relying on offline operations. *Devices* focus on in-transit data management, delegating offline operations to higher layers.

Chapter 4

Programmable I/O Processors

In this Chapter, we introduce a framework for building *I/O Processors* for in-transit processing and data distribution. The framework distinguishes the data plane from the control plane. Using the provided declarative language the developers can model the desired I/O path as a directed acyclic graph of processing modules. The framework automates the synthesis and produces a running instance of the target *Processor*.

4.1 Objectives

An *I/O Processor* is an intermediate that allows the injection and the execution of data processing routines without the burden of systems programming. A key different between *I/O Processors* and *Devices* is that the former operates on data-level (e.g., transformations, routing, stripping) whereas the latter on request-level (e.g., aggregation, chunking). Next, we present a few of the challenges *Processor* are trying to address.

4.1.1 In-transit processing

It is rare for application data to persist in the same format as derived from the computation. Typically, data-stores apply certain transformations such as compression, deduplication, or encryption before persisting data on physical media. Although such functionality saves the application developers from mixing I/O processing logic with computation logic, it locks the application to data-stores that support these features.

Interposing the application and the storage *Devices*, *I/O Processors* make it possible to apply post-processing routines to application output, transparently to the application. Suitable tasks include those that reduce data without loss of scientific validity (e.g., compression, deduplication), improve information content (e.g., add simulation timestamps), generate metadata (e.g., indexing), or perform lightweight analysis for visualization dashboards.

4.1.2 Data Distribution

For a client to transfer data from and to a *Device* is a rather straight-forward process. Where it gets significantly more complicated is when a client needs to leverage more than one *Devices* simultaneously. Reasons include performance, resiliency, or privacy.

One way to achieve data distribution to multiple independent *Devices*, or platforms, is to incorporate such a mechanism into the source code of the client application. Unfortunately, parallel programming is notoriously difficult and comes with a boilerplate code for thread execution, bridging (e.g., buffering), and synchronization for thread safety. Frameworks and languages such as MPI-IO library [51] or Chapel [114] can address some of the above, but have a steep learning curve and therefore are not suitable for our purposes. Moreover, every new application must reimplement the same codepaths.

In our proposal, all that it takes from the developer is to describe the desired I/O logic in a graph-based language and the *I/O Processor* handles the rest.

4.1.3 Bidirectional streams

In the literature one can find a vast amount of stream processing engines for such as Flink [25] and Spark [134] BigData, or GNURadio [16] for signal processing. Similarly, graph-based programming dominate domains such as distributed computation [37, 71], parallel task scheduling [28, 134], and machine learning [1].

In general, all of the above exhibit the same property: data flow in one directional. The flow for I/O usage though is bi-directional since it does not suffice to only process data on write, must data must be unambiguously reconstructed to their original form on reading. To this end, *I/O Processors* reverse the concept of "a single streaming network to serve multiple events" towards "every event owns its streaming network".

4.2 Driver Synthesis

Our framework, called Transparent Regulated I/O (TrIO), aims at designing *Processor* drivers as directed acyclic graphs of processing components [90]. Presenting this type of driver as a graph, instead of a stack, makes it more natural to express various distribution schemes. One can imagine the difficulties of expressing as a stack the statement “mirror data in two streams, encrypt the first, and compress the second”. With a graph, the topology seems as natural as the statement itself. The outline of a Processor driver is depicted in Figure 4.1. It consist of graph definitions for upstream and downstream flows, and wrappers that pipeline the graph outputs with Devices or other Processors for further processing. The latter makes it possible to federate *Processors* that reside on different nodes into a *Composite Network*, and move I/O processing logic from computational nodes to filtering nodes, closer to the data. That comes in handy for computational nodes with exhausted memory or running memory-intensive applications. It is

also useful for nodes equipped with expensive hardware that cannot waste for simple filtering tasks.

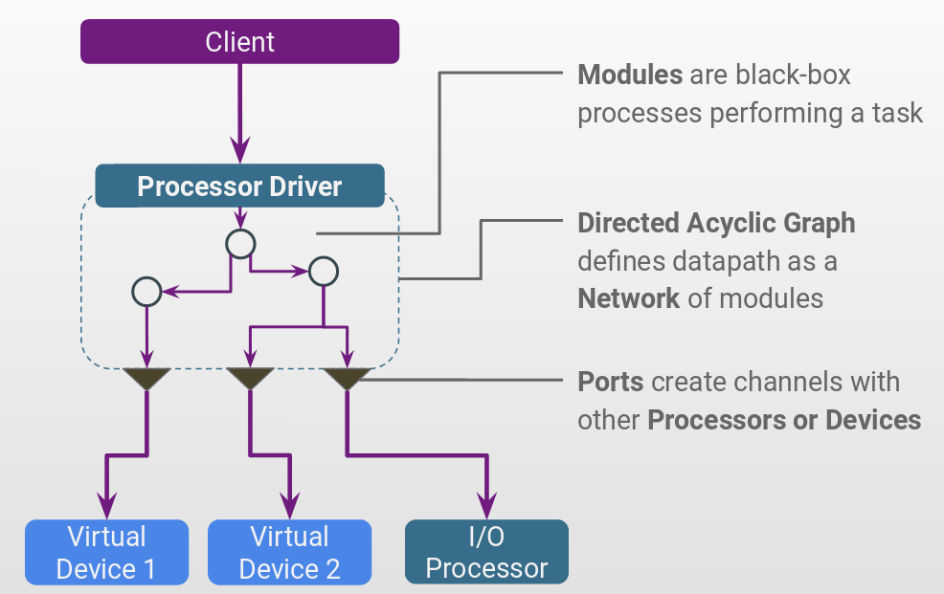

Figure 4.1 – Synthesis Service for Processor drivers. The driver abstracts the processing path as a directed acyclic graph the outputs of which can be forwarded to Devices or to other Processors for further processing. The developer models the driver as a graph of processing components, as shown in Listing 4.4

TrIO adheres to a visual programming paradigm called flow-based programming (FBP) [88]. The paradigm, invented by J. Paul Morrison in the early 1970s for banking applications, relies on bounded buffers, named ports, information packets with defined lifetimes, and separate definition of connections. FBP views an application not as a single sequential process with a discrete start and finish points in time, but as a network of asynchronous “black box” processes. In this view, the focus is on the application data and the applied transformations to produce the desired output.

The major characteristics of Flow-based programming paradigm are [91] :

- Unified approach in software development from domain analysis to coding. Start with diagrams and data flows and continue the structural breakdown until the processing algorithms. Essentially it maps hardware engineering practices on software development
- Concurrency comes by design, not by purpose. Graph nodes run in parallel and share state by communication. FBP applications run in less elapsed time than conventional programs, and make optimal use of all the processors on a machine, with no special programming required to achieve this.

Known appliances of FBP programming include Data analysis and ETL (e.g., Pentaho Kettle, Pypes), Multimedia broadcasting (e.g., Kamaelia), Simulations (LabVIEW is flow-based), Networking (AMQP is a similar approach). The *scheduler* is a piece of software responsible for

interpreting the abstract graph into an object running in memory. The basis of our scheduler is GoFlow [124] implementation of FBP, adapted for I/O purposes in a distributed setup.

4.2.1 Processing kernels

Kernels are fundamental event-driven processing components. They communicate using fixed-capacity connections that convey structured data chunks, called "Information Packets" (IP). Connections are attached to a kernel through a port; an agreed name between the kernel and the graph definition. The separate definition of connection minimizes kernel interdependence and skips the need to integrate a routing table within the kernel scope. Kernel "multiplex" incoming requests, passing each parent request on to one (processor/compress), some (processor/stripe), or all (processor/mirror) of the output ports. In case that multiple outputs connect to a single input, the packets arrive in FIFO order. An output port can connect only to one input port; otherwise, the behavior is undetermined.

All the kernels must include three public fields named "Ingress", "Egress", and "Runtime". The first is the input ports, the second the output ports, and the third a context that we will explain later (Figure 4.1). Given those fields, the scheduler automates the kernel integration (e.g., fixing pointers) without further involvement from the developer. Both input and output ports can be scalar or array-type (e.g., for stripping). The number of ports per kernel is not limited, but it is recommended to keep them a few, otherwise, kernel's cohesion and logical consistency suffers.

```
1 type Kernel struct {
2     Runtime process.Environment
3     Ingress  <-chan process.Packet
4     Egress chan <- process.Packet
5 }
```

Listing 4.1 – All handlers must include these three fields. The scheduler use them to add the kernel into the pipeline

Event handlers When an *Information Packet* arrives at one of the input ports, the scheduler spawns a new thread to handle it. The called method is the one named after the port identifier. Because asynchronous threads can lead to race conditions and unexpected program behavior if two threads try to modify the kernel state, *Processors* have a built-in mechanism to make state modification thread-safe. On presence, the scheduler will invoke StateLock() method before calling an event handler and StateUnlock() after event handler returns. The scheduler can also handle the uncontrollable flow of incoming requests that can cause a "fork bomb", i.e., reach the maximum entries in the process table or exhausted RAM. It includes a Thread pool mechanism that provides a Pool of workers in which every process runs a specific number of workers which compete for input and process it concurrently.

Data exchange Devising a proper data exchange schema proved to be more challenging than expected. The operating systems impose certain limitations on the maximum size of a conveyed Information Packet (IP). That renders it impossible to transfer a large sequence of bytes at once. Dividing the content into multiple IPs will cause the kernel to spawn new handlers for each received IP. Implementing complex multiplexing mechanisms within the kernel makes it more difficult to reason about the state and take snapshots.

Our solution leverage Golang's [29] pipes [72] that provide streaming methods to buffers. Instead of moving data around, we move reading and writing edges of the pipe. At any point in time, a buffer is "owned" by a single kernel. What happens with the pipes is that a kernel "leases" the buffer to next kernel for reading or writing. When one of the edges close the pipe, the termination signal propagates to the other edge and acts accordingly. For example, if a previous kernel closes the pipe, the successive kernel becomes aware that no data will follow and can gracefully terminate. Conclusively, channels connect kernels, information packets cause kernels to spawn handlers, and handlers exchange data via pipes or streams.

Bidirection The I/O process is bidirectional. The direction from clients to the storage called upstream, and the direction from storage to clients called downstream. For the client to interpret the data correctly, the downstream must inverse the transformations occurred in the upstream. If the kernels participating in the upstream are stateful, then their state on the upstream is also needed to reconstruct the original data. *TrIO* provides an environmental variable for *kernels* to store and load their state, in a key-value manner.

On upstream a kernel K creates a pipe and forwards the reading edge to the next kernel N. Kernel K start pushing the process results to the writing edge of the pipe. Kernel N can retrieve the results by the reading edge of the pipe. When kernel K has pushed all the data, closes the pipe and stores the number of pushed data to the database. The pipeline gracefully closes top-bottom, starting from the application. On downstream, kernel K load the state and allocated a buffer equal to the number of written data. It then creates a pipe and forwards the writing edge of kernel N for populating the buffer with data. The pipeline gracefully terminates bottom-up starting from the *Devices*.

Codes 4.2 and 4.3 exemplify how kernels can use pipes and the environmental variable, for upstream and downstream, respectively.

Available Kernels

- **Encryption**: is the process of encoding a message or information in such a way that **only authorized parties can access it** and those who are not authorized cannot. We base our implementation on Advanced Encryption Standard (AES) [27]. AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data. The Output Feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XORed with the plaintext blocks to get the

```
// Upstream
func (k *Kernel) OnIngress(packet process.Packet) {
    pr, pw := io.Pipe()

    defer pr.Close()

    k.Egress <- process.Packet {
        Reader: pr,
    }

    /* Transformations go here. For the demo we copy */

    n, err := io.Copy(pw, packet.Reader)
    if err != nil {
        k.Runtime.Abort(err)
    }else {
        k.Runtime.Store("written", n)
    }
}
```

Listing 4.2 – Illustration of upstream pipelining. The kernel is passing down to its outputs the reading edge of a local pipe. In this example, we copy directly to input to output. In more advanced scenarios the kernel would keep a local buffer with the encoded data. On success, the kernel record its state in a provided key-value map

ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. Each operation on output feedback block cipher depends on all previous ones, and so encryption is sequential.

- **Compression**: reduces the data size by determining if a sequence of bits that define any particular piece of data can reduce to a smaller sequence. Zstandard [32] is a fast lossless compression algorithm introduced by Facebook, targeting at real-time compression. Zstd can also offer stronger compression ratios at the cost of compression speed. The tradeoff between speed and compression is configurable by small increments. The smaller the amount of data to compress, the more difficult it is to compress.

- **Deduplication**: refers to a specialized data compression technique that eliminates duplicate blocks of repeating data before being written to a storage device [83]. Unlike simple encoding schemes, like Base-64, where every three contiguous input bytes corresponds, in order, to 4 contiguous output bytes, **in compression the input relates to the output in a sophisticated way**. The implications are as follows i) users cannot merely take compressed data and decompress an arbitrary portion of it, such as “decompress the last 500 bytes of this file”. They need to read the entire compressed file from the beginning or at least start from some well-known point in the stream. ii) Modification of the uncompressed input may have arbitrarily large impacts on the compressed output. For example, changing

```
// Downstream
func (k *Kernel) OnIngress(packet process.Packet)
    pr, pw := io.Pipe()

    defer in.Close()

    k.Out <- process.Packet{
        Writer: pw,
        Info: struct{}{},
    }

    n, err := k.Runtime.Load("written")
    if err != nil {
        u.Runtime.Abort(err)
    }

    /* Transformations go here. For the demo we copy */

    rb, err := io.CopyN(packet.Writer, pr, n)
    switch {
        case err != nil:
            k.Runtime.Abort(err)
        case rb != n:
            k.Runtime.Abort(errors.Corrupted(n, rb))
    }
}
```

Listing 4.3 – Illustration of downstream pipelining. The kernel is passing down to its outputs the writing edge of a local pipe. Before doing anything else, it first loads the state. In this example, we copy directly. In more advanced scenarios the kernel would keep a local buffer with the decoded data

a single-byte in the input may change every subsequent byte in the output. Deduplication works in a way that **random access into a file segment is still possible** albeit the file is in compressed form, e.g., by having an index such that the location of any byte can be located. The tradeoff for this convenience is that deduplication only works with large blocks, or the cost of tracking each block becomes prohibitive. Although deduplication was initially developers as a way to improve storage efficiency for backups, with the advent of flash storage, it became a way to reduce the amount of data written to the writes so to reduce driver wearing.

- **Mirroring**: is the real-time replication of data onto separate physical location so to ensure continuous availability. Replicating data over long distant location allows to failover and use a standby copy of the data, hence enabling **fault-tolerance - continuous availability with minimal user interruption**. Replication also helps with load-balancing and performance.

Clients can fetch in parallel chunks from mirrored data in order to minimize overall operation time. For geographically dispersed locations clients can choose to fetch data from the replica **closer to their proximity, thus achieving lower latency and higher throughput**. Due to real-time data copying, the information stored from the original location is always an exact copy of the data from the production device. Hence, there are no induced consistency issues derive from version skewing between the replicas as occur in most eventually consistent systems.

- **Stripping**: is the technique of segmenting logically sequential data, such as a file, so that consecutive segments are stored on different physical storage devices. Striping allows the minor data access throughput of each storage devices to be cumulatively multiplied by the number of storage devices employed. It increases the total data throughput by spreading segments across multiple devices, in parallel. Apart from parallelization and bottleneck mitigation, disbanding incoming data also a useful method for **balancing I/O load across an array of devices**. One method of striping is done by interleaving sequential segments on storage devices in a round-robin fashion from the beginning of the data sequence. This works well for streaming data, but subsequent random accesses will require knowledge of which device contains the data. If the data is stored such that the physical address of each data segment is assigned a 1-to-1 mapping to a particular device, the device to access each segment requested can be calculated from the address without knowing the offset of the data within the full sequence. Because different segments of data are kept on different storage devices, the failure of one device causes the corruption of the full data sequence. In effect, the failure rate of the array of storage devices is equal to the sum of the failure rate of each storage device.

  zasw4

- **Erasure Coding**: overcomes data availability and safety disadvantages of stripping by enriching the original content with redundancy data and spread it across devices. A subset of this data is enough to regenerate the original data, even if parts of the original content have been lost. The most established erasure-coding algorithm invented by Reed and Solomon [102]. Reed-Solomon algorithm, as it is widely known, takes a message, breaks it into n pieces, adds k “parity” pieces, and then reconstruct the original from n of the (n+k) pieces. Erasure coding allows setting arbitrary ratios of original data and coding data. With a ratio of m parts of original data to n parts of coding data, the code can tolerate the loss of any n parts and regenerate the original m parts. For example code of m:n = 8:3 enriches every eight parts of data with three parts of coding data and spreads the data across 11 (8 + 3) disks. This encoding can then tolerate the loss of any three disks and generates a redundancy blow-up of just (m + n)/n = (8 + 3)/8 = 1.375. **Compare this with three-way quorum replication, which can tolerate the loss of only 1 or 2 disks and has a blow-up of a factor of 3.** Erasure coding performs best in cases of sequential data writes as it computes the coding parts on the fly and writes them along. If written in random order, write

performance for data degrades severely. The kernel needs to read all data of the coding group first, recompute the coding parts, and then write out the modified original data along with the coding data. That amplifies any random write by m – 1 read and m writes. As erasure coding stores the original data as-is, reading erasure coded data behaves just **like reading replicated data**. Thereby, erasure coding is suggested as a mean for redundancy for sequentially written data. For everything else use replication [128].

- **Windows** are specialized kernels that communicate with external systems. Contrary to the previous kernels whose inputs and outputs reside entirely within the *Processor*, windows interact with other systems. Use cases include the augmentation of the in-transit bytestream with data from external sources (e.g., simulation timestamps [18]) and the export of data summaries for real-time visualization (e.g., like tensorboard [43]).

4.2.2 Graph

The vertices in a *Graph* represent the components, and the edges represent their connection. *Graphs* supports multi-level hierarchies with components being either kernels or nested graphs. The *Network* is an in-memory instance of a top-level graph. To realize the abstract *Graph* into a running instance, the scheduler allocates an empty *Network* that populate with components and connections, at runtime, as it walks the graph. Thanks to data coupled components -the loosest form of coupling- the scheduler can turn off and on components without affecting the rest of the *Network*. Another way to perceive the process is as “compiling” the graph. Because the allocation and compilation are relatively expensive operations, when a *Network* closes we do not destruct it. We reset its state and place it into a *Network* pool. Future requests can retrieve only of the existing instances instead of compiling a new one.

The following directive adds a new component to the *Network*

```
network.Add(new(ComponentType), "processName", new(Environment))
```

The first argument is the actual process object. The second is a string name which will be used to reference the process within the *Network*. The reference names do not have any particular meaning other than identifying components within the graph. Graph developer is responsible for ensuring the uniqueness of assigned identifiers, or the expected behavior is undetermined. The third argument is optional and is used to provide the component with a key-value map for state keeping.

The following directive links two components in the *Network*

```
network.Connect("senderProc", "receiverProc")
```

The first argument is the reference name of the sending kernel and the second argument the reference name of the receiving kernel. A thing to notice is that there is no explicit definition as to which input and output ports to connect. That drifts from the actual FBP definition. The first

reason is for simplicity; the scheduler automatically resolves the port index in a first-come-first-served order. The second reason is to avoid loops (feed output back to input) and cycles (feed output back to the input through a proxy) that can lead to livelock situations.

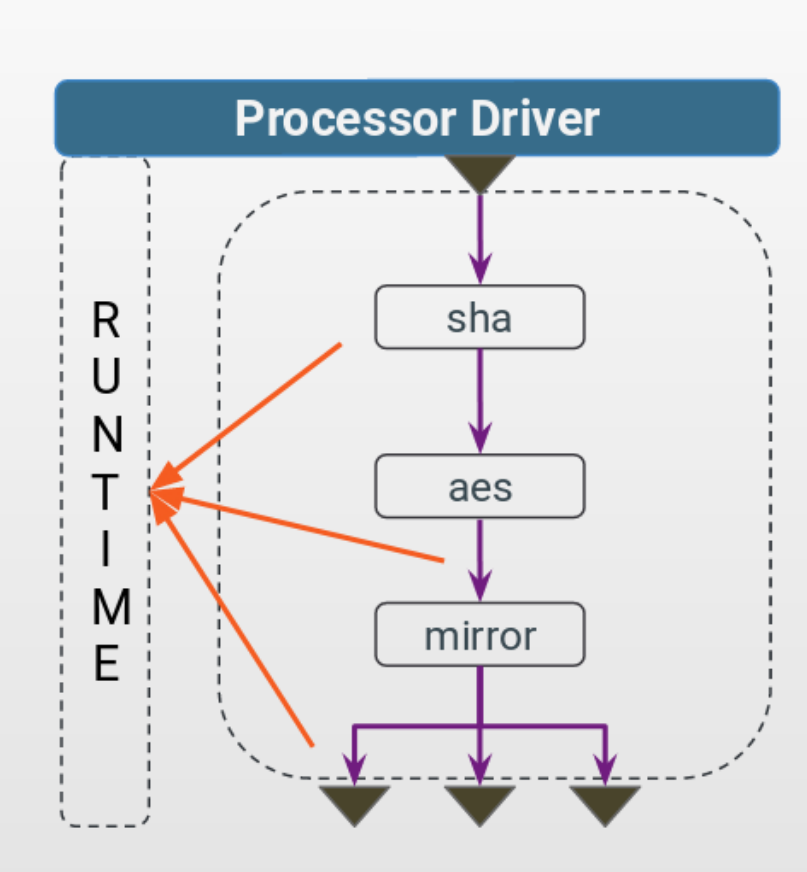

Figure 4.2 – The Processor runtime consumes a TrIO graph and produces the respective network (graph instance). It case of Macro-graphs, the runtime also provides a lightweight key-value map for saving the evaluted expressions. This is needed so that the downstream can reverse the upstream graph and correctly interpret the written data

Macros Programming in graphs is not a panacea as it involves predetermined and hard-coded paths. That translates to a large collection of graphs to maintain. For example, for a replication scenario with a mirroring component, the administrators must maintain a different graph for every different number of replicas – different topology, different graph.

To mitigate the reusability issue, we employ *Macro Conditions*. As the scheduler walks the graph it evaluates the conditions and decides which branch to follow, or how many outputs to add to a kernel. The conditions can be user-defined (e.g., number of replicas) or system-defined (e.g., CPU-load). By deferring the path selection and taking into account the system state, the scheduler can reroute I/O for load balancing, perform adaptive optimization such as avoid previously failed paths [125] or dynamically switch I/O processing kernels. Code 4.4 shows a snippet of a graph augmented with *Macros* for disabling compression if the CPU-load exceeds

80%. Our technique reminisces just in time compilation (JIT) [11] used in interpreted language to improve the application performance at runtime. The tradeoff for this dynamism is the need to recompile a new *Network* for every new request. Further, the downstream graph must know the selected paths on the upstream so to interpret the data correctly, i.e., it should know if data were written in the compressed form and decide whether to include a decompression component or not.

Overall, static *Graphs* perform generally faster because compiled-once-run-many-times. It is also possible to perform certain optimization (e.g., block size) since all the components are known a prior. Oppositely, dynamic *Graphs* are more flexible, more powerful, but require compilation for every request.

```
func (d *Driver) NewUpstream() base.Upstream {
    // Allocate a basic process network
    upstream := base.NewUpstream()

    upstream.Add(dedup.Uplink{}, "dedup")
    upstream.Add(forwarder.Uplink{}, "holder")
    upstream.Add(strip.Uplink{Stripe:d.slen}), "strip"

    upstream.MapInPort("In", "dedup", "In", nil)
    upstream.Connect("dedup", "holder")

    if d.Runtime.CPULoad < 80 {
        upstream.Record("cpu", d.Runtime.CPULoad)
        upstream.Connect("holder", "compress")
        upstream.Connect("compress", "strip")
    } else {
        upstream.Connect("holder", "strip")
    }
    upstream.MapOutPort("strip", nil)
    upstream.MapOutPort("strip", nil)

    return upstream
}
```

Listing 4.4 – Composition of customized Processor driver (as graph) in the TrIO language. For brevity, we only include the graph definition for the upstream. The Macro evaluation allows to switch on and off the compression according to the system load. The upstream persistently record the conditional variable so that downstream graph can reconstruct the path and interpret the data correctly

Downstream is not Upstream Previously we argued that a downstream graph must inverse the process occurred in the upstream graph in order to correctly interpret the data. The subtle point is that inversion does not necessarily imply symmetrical graphs. For example, an upstream graph

with a mirroring kernel can be represented on the downstream by (i) a graph with a decision-making kernel for reading from one replica or another (ii) a graph with a kernel that simultaneously reads both replicas (iii) a graph that completely neglects the second replica (e.g., when it is archived). Hence, two graphs that logically inverse one another may differ both in kernels and topology.

A related challenge we had to face in order to support bidirectionality was the topology of the graph. The downstream graph is flipped compared to the upstream. The *Device* that was the destination on upstream is the data source for downstream. An inversion is relatively simple for a topology with serial pipelines, but it becomes significantly more complicated if the topology involves branches or multiplexers (e.g., for stripping). Our solution was to invert the data transfer logic instead of inverting the whole graph. To do the "trick", we employ the concepts of pipes and key-value maps as mentioned previously.

4.3 Driver Runtime

The *Network* as described so far is a stream-processing engine. In order to start processing data, someone must feed data to the engine input. Similarly, someone must consume its output. Aiming at reusability and simplicity, we deliberately kept the data ingestion and digestion mechanisms separate to the *Network*.

4.3.1 Network Ports

In contrast to the ports of components which are physical structures with public fields, the *Network* ports are virtual mappings for streaming data from external processes to the *Network*. They are also a boundary that hides the internal *Network* structure from the outer world.

Network input ports are declared in the *Graph* using the method

```
network.MapInPort("processName", func([]string), func(<-chan error) error)
```

The first argument is the reference name of the component used similarly to Connection directive. The second argument is a callback function for resolving compatibility linking. One of the problems of existing storage APIs is that they do not explicitly advertise supported features within the system. They may be publicly available in human-readable documentation, but the lack of machine-to-machine information makes it impossible to automatically resolve whether the system implements a certain functionality or not. For this reasons, network input ports are annotated so to detect "compatibility" with the other edge, whether that be another *Graph* or a *Device*. Typically, systems resolve compatibility in a top-bottom manner. The upper edge (e.g., client) resolves whether a lower edge (e.g., server) qualifies to the requirements and decides to connect or not. In our bottom-up approach, the upper edge sends a list to the lower edge, and the lower edge replies whether it meets the requirements or not. The upper edge is the output

network port and the lower edge the input network port of a successive graph or a *Device*. The third argument is a callback for deciding the returned error code of the *Network* based on the outcome of the output ports. For the majority of distribution schemes, all the outputs must be written successfully before declaring the top-level operation as successful, but it not also always the case. For example, in "at least" replications schemes the operation is successful when K out of N outputs (k <= N) written successfully. In such schemes, the operation can continue even in the presence of a limited number of errors, reducing the overhead of going into the recovery process. Further, it allows to acknowledge an operation faster and still be confident about the resiliency guarantees. The client-side equivalent is SetInPort().

Network output ports are declared in the *Graph* using the method

```
network.MapOutPort("processName", new(Context), ...requirements)
```

The first argument is the reference name of the component used similarly to Connection directive. The second argument is a Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes. It is necessary for ports connected with other entities over the network. On communication lose, the operation will indefinitely block until communication recovery. Contexts trigger an error code after a timeout and *TrIO*, which appear to the rest of the system as if the operation had failed. *TrIO* is responsible for conveying the error code to the input port for the final error code decision, as previously described. The third argument denotes the requirements for connecting to another entity. The tradeoff for the relaxed security is the ability to pass hints to the other edge. Hints can be advisory in order to avoid implementing the same functionality twice on different layers (e.g., deduplication), or mandatory and denote a functional requirement that cannot be missing without jeopardizing application correctness (e.g., encryption). Hints are also useful in the *Device* selection process. The client-side equivalent is SetOutPort().

4.3.2 Federated Network Discovery

In order for *Tromos* to support atomic operation it is necessary to resolve the number of affected *Devices* deterministically. For *Processors* whose outputs are linked directly to *Devices* it is straightforward. Affected devices are those connected to the *Network* ports. It not trivial though when *Processor* combine to form a *Federated Composite Network*. That is because *Processors* can link in variable and dynamic ways and there is no global graph to describe the structure of the *Composite Network*. For example, a client sends its data to the input of *Network A*, whose outputs connect to the inputs of *Networks B and C*. A client may ask *Network A* for its outputs, but will it has no information about the *Devices*. The problem is similar to a client asking a DNS server for a name. If the server has no record information, either it will ask an affiliated server and act as a recursive middleman, or directly reply with the affiliated server and let the client do the next iteration.

To resolve the affected *Devices*, we augment *Processors* with a discovery and resource reservation protocol (RSVP) [135]. Using that protocol, clients can establish end-to-end channels with *Devices* regardless of the number, the location, and the graph structure of intermediate *Processors*.

RSVP protocol states that:

i. when a *Processor* receives an RSVP request it instantiates the abstract graph to a *Network* object and broadcasts the RSVP to all the output ports of the *Network*

ii. the "leaf" *Processors* whose outputs do not link to other *Processors* or *Devices*, acknowledges the request and propagate backward an RSVP reply with its identifier

iii. Once "root" *Processor* whose inputs do not connect to other *Processor*, receives RSVP replies from all its outputs, it establishes the channel

iv. subsequent I/O requests traverse through the established channel as distinct streams.

Service API	Comment
Load(Plugin) (Processor, error)	Load a datapath plugin
Processor.Discover() error	Discover the composite datapath
Processor.NewUpstream() (Upstream, error)	Spawn a new upstream network thread
Processor.NewDownstream() (Downstream, error)	Spawn a new downstream network thread
Graph Developer API	**Comment**
Up.Add(new(component), new(pname)) error	Add a new component to the network
Up.Connect(pname, pname) error	Link components
Up.Record(new(key), new(value)) error	Store state in a key-value map
Up.MapInPort(pname) error	Bind input port to an internal component
Up.MapOutPort(pname) error	Bind internal component to output port
Application API	**Comment**
Up.SetInPort(new(channel)) error	Bind top-level channel to network port
Up.SetOutPort(new(channel)) error	Bind top-level channel to network port
Up.SetRuntime(env Environment) error	Environment for storing the state
Up.Init() error	Start running the network
Up.Wait() error	Wait until the network terminates

Table 4.1 – A simplified version of the Processor API. For convenience to the reader, we omit some arguments.

4.4 Summary

In this Chapter we present a framework for composing *Drivers* for I/O *Processors*. The *TrIO* is a declarative language that abstracts *Drivers* as a graph of basic processing elements called *kernels*. *Graphs* may also contain *Macros* conditions that can manipulate the final graph structure

at runtime. The arguments for the macro evaluation can be user-defined or system-defined. The scheduler is an engine for building a process *Network* according to the graph definition. The *Network* is a graph instance running in memory, whose outputs whose outputs can link to other *Processors* or *Devices*.

When linking a *Processor* to another *Processor*, it creates a conceptual *Composite Network* with dynamic and unknown topology. In order to resolve the topology of that *Composite Network* we employ a discovery and reservation protocol called *RSVP*. That makes it possible to create end-to-end channels between clients and *Devices*, regardless of the number of intermediate *Processor* and whether they run on the same or on different hosts.

4.4.1 Related Work

In data centers, the IO path to storage is long and complicated. An IO request from an application to distributed storage traverses not only the network but also several software stages with diverse functionality with opaque interfaces between them. Stages include caches, hypervisors, IO schedulers, file systems, and device drivers. This set of ordered stages is known as the storage or IO stack. That makes it hard to enforce end-to-end policies that dictate a storage IO flow's performance (e.g., guarantee a tenant's IO bandwidth) and routing (e.g., route an untrusted VM's traffic through a sanitization middlebox). These policies require IO differentiation along the flow path and global visibility at the control plane. Software-Defined Network (SDN) [82, 87] and Software-Defined Storage (SDS) are centralized services for policy-based provisioning and hardware-independent management. By decoupling the control plane from the data plane, SDN allows for packet-processing routines to be injected into the switching infrastructure and create an overlay network without the limitations of traditional networks. Similarly for SDS:

The seminal study IOFlow [121] proposed an architecture that uses a logically centralized control plane to enable high-level flow policies. IOFlow adds a queuing abstraction at data-plane stages and exposes this to the controller. The controller can then translate policies into queuing rules at individual stages. While packet routing is fundamental to networks, no notion of IO routing exists on the storage stack. The path of an IO to an endpoint is predetermined and hard-coded. That forces IO with different needs (e.g., requiring different caching or replica selection) to flow through a one-size-fits-all IO stack structure, resulting in an ossified IO stack. Although IOFlow also made a case for request routing it only explored the concept for bypassing stages along the path and did not consider the full IO routing spectrum where the path and endpoint can also change. sRoute [116] proposed a full routing abstraction for the storage stack, built upon IOFlow and borrowing two specific primitives: classification and rate limiting based on IO headers for quiescing. A key strength of such architecture is that it works with unmodified applications and VMs.

Data Service [2, 3] presented some higher level I/O abstractions for carrying out data processing such as transformation, reduction and scheduled storage for asynchronous flushing of data to the disk via the file system. It manages output data from 'source to sink': where/when

it is captured, transported towards storage, and filtered or manipulated by service functions to improve its information content. That results in runtime flexibility not only in which services to employ, but also in selecting nodes for data staging and how they use I/O resources. In principle, that services directly leverage EVpath [31]; an event transport middleware layer designed to allow for the smooth implementation of overlay networks, with active data processing, routing, and management at all points in the overlay.

Multipath I/O [125] is the ability to provide increased performance and fault tolerant access to a device by addressing it through more than one path. For storage devices, Linux has seen several solutions that were of two types: high-level approaches that live above the I/O scheduler (BIO mappers), and low-level subsystem specific approaches. Each type of implementation has its advantage because of the position in the storage stack in which it has been implemented. The authors focus on a solution that attempts to reap the benefits of each type of solution by moving the kernel's current multipath layer below the I/O scheduler and above the hardware-specific subsystem.

Chapter 5

Programmable Coordinators

In this Chapter, we introduce a framework for building *Coordinators* for namespace management, metadata management, and distributed synchronization. The framework distinguishes the data plane from the control plane. Using the provided declarative language the developers can model the device properties as a stack of basic components. The framework automates the synthesis and produces a running instance of the target *Coordinator*.

5.1 Objectives

In the previous Chapters we saw *Devices* for unifying various storage platforms and *I/O Processors* for in-transit processing. The combination of the two consist an end-to-end datapath that span from client application to the devices. To this end, the datapath is about a client sending data to *Devices*. The *Coordinators* are about synchronizing multiple clients sending data to *Devices*. Using composable drivers, the *Developers* can customize *Coordinator* semantics according to the application requirements, without the direct input or any management from the application. Existing drivers serialize access to the log, monitor requests in real time, and augment upper layers with access-pattern information. Next, we present a few of the challenges *Coordinators* are trying to address.

5.1.1 Metadata Catalog

Coordinators provide a key-value database for cataloging and discovery of logical files. Logical files do not contain data. A logical file is a view or representation of one or more physical files. Because *Coordinators* leverage existing databases to durably persist the catalog they face, and address, similar challenges to *Devices*. Every database is trying to address a different problem with every problem family asking for a different combination of data structures and hardware. They may use Btrees for storage efficiency and high sequential throughput on hard disks (e.g., PostgreSQL); Btree+ for read-intensive workloads as they minimize I/O amplification and favor

range iteration (e.g., LMDB, BoltDB [17]); or LSM for high insert volume as it leverages random-access of SSDs [57, 73]. They may also integrate optimizations for network-access [74], or trade off correctness (e.g., ACID) for concurrency (e.g., MVCC). Thereby, no database can perform optimally to all workloads. The best-fit selection depends on the application characteristics and the available hardware on the host.

5.1.2 Distributed Synchronization

A multi-storage solution involves multiple data-stores with potentially different semantics. Even in the rare case that these data-stores individually support transactions it does not suffice to guarantee the correctness of a top-level operation. A data-store transaction guarantees ACID properties during transfer and persistence time on the particular data-store. It does not concern about the overall operation status. In contrast, a multi-storage transaction encapsulates full-stack information such as data-store level operations, interposing I/O processing, top-level access information (e.g., offset in the logical file), and so on.

Coordinators provide a set of distributed synchronization primitives that allows developers to focus on core application logic without worrying about the distributed nature of the application. Depending on the physical problem an application tries to solve, it may opt for strict ordering and up-to-date information or stale information in favor of concurrency and scalability. Typically, applications either delegate the consistency semantics to the general purpose or use the primitives of generic distributed synchronization engines. The pitfall in the first approach is that provided semantics may not precisely match the application requirements. For example, metadata intensive workloads are likely to bottleneck at the filesystem control plane due to namespace synchronization that comes with lock contention on directories, transaction serialization, and RPC overheads.

In contrast, the second approach gives the flexibility to use selectively API calls that best meet the desired behavior. For example, users should call “sync” API to synchronize and read up-to-date values or “async” API to read immediately stale values. *Coordinator* framework provides a single API set for which an externally composed driver govern the semantics. Thereby, an application can switch from strong consistency to relaxed consistency without changes into the source code. The only, optional, call that must be hardcoded on the client-side is about acquiring access pattern information. More specifically, clients can ask *Coordinators* about the trends of a key (e.g., access frequency, most frequent client) in order to make intelligent data placement decisions. Such flexibility though comes with the cost of an additional request over the network which is not always needed.

5.2 Driver Synthesis

A *Coordinator* driver is a stack of pipelined synchronous handlers. As synchronous we refer to the property that a handler does not reply to its caller until the lower handlers have replied first.

That is to ensure that once an operation is acknowledged, it has been indeed executed and not just queued. As a result, neither asynchronous stacks, as in *Device* drivers, nor graphs, as in *Processor* drivers, can adequately provide the above functionality. We design the *Coordinator* API around the abstraction of a shared log that can be accessed concurrently by multiple clients over the network. An agreed global ordering reduces to the ordering of events in the log [12,104]. Based on the above, deciding the consistency level becomes a manner to choose whether to include a sequencer on the driver. Code: 5.1 presents a skeleton of a sequencer handler that serializes access to the lower layers.

```
type Sequencer struct {
     // Used by synthesis service to pipeline the handler
    coordinator.Stack
    // Some internal structures
    writes cmap.ConcurrentMap
}

func (s *Sequencer) CreateOrReset(key string) error {
    // Step 0: pre-process (lock the key)
    once := utils.Once()
    s.writes.Upsert(key, make(chan struct{}),
        addSignal(once))

    // Step 1: forward request to the next layer
    err := s.Stackable.CreateOrReset(key)

    // Step 2: Post process
    s.writes.RemoveCb(key, closeSignal(once))

    // Step 4: Return atomic operation status
    return nil
}
```

Listing 5.1 – Coordinator Handler Definition. All handlers must include a public coordinator.Stack field. The synthesis software uses it to pipeline the handler. The other important feature is that handlers are synchronous; they do not reply until the lower handlers have replied first

5.2.1 Modeling Language

The *Keyzone* is a YAML-compatible language for composing drivers for *Coordinators*. It consists of a unique identifier that denotes the range of keys for which the specific *Coordinator* is responsible for, functional information such credentials and paths, and a stack of synchronous handlers (Code 5.2). Handlers, or stack layers, are called by parents and in turn call children. Some layers are “final” as they terminate requests instead of passing them on, so they do not have children. For example, coordinator/proxy/client forwards requests to another node through

the network, coordinator/connector/boltdb invokes database operations. Others are "initial" and inject requests into the system from elsewhere, so they do not have parents. For example coordinator/proxy/server injects requests from the network. Intermediate layers have one parent and one child. Layers must implement the same interface in order to be stackable, but this does not mean that all layers are interchangeable. Depending on their position in the stack we differentiate handlers to the following categories:

According to their position in the stack, we differentiate handlers to the following categories. Figure 5.1 depicts the synthesis process and Figure 5.2 presents a snippet of a *Coordinator* definition.

Connector

Connectors are the lowermost layers in the stack. They are "bookkeepers" running on top of third-party databases. Keys in the books are tuples of ledgers that store intention records for uncommitted transactions and update records for committed transactions [6, 104]. On commit, the connector updates the entries on the two ledgers atomically. As ledgers we refer to shared logs meant to store system information such as transactions and metadata, not user-data. As the authors of Corfu [12] state:

> If we can construct a (suitably performant) linearizable shared log, then this primitive can be used as a key building block to solve some hard distributed systems problems: an agreed upon global ordering becomes simply the order of events in the log.

Inspired by that, *Connectors* speculatively execute transactions by appending them to the shared log and then use the log order to decide commit/abort status. Available connectors include:

- **BoltDB**: An embedded key/value database optimized for read operations [17]. It performs optimally for workloads with demands on scanning and key iteration.
- **Badger**: An embedded key/value database optimized for use in SSDs [57]. It performs optimally for write-intensive workloads.

Translators

The *Translators* are layers handlers that control access to the shared log. They can decide whether to forward, block, or immediately reject a request. They can also access the request payload but cannot modify it, e.g., to visualize content. The *Translators* are volatile and their state resides entirely in memory. In order to guarantee correctness in case of a crash, they do not acknowledge requests before all the lower layers have acknowledged it first. Recursively, that ensures that a request is persisted (in the ledger) before the *Coordinator* replies to the client. In case of a crash, the impact to the system will be the same as if the network ceased the request. Available translators include:

- Visualizer: plotting real-time request information (e.g., heatmaps for afffected segments)
- Monitor: exposes access-pattern information to the *Device* selector, such as frequency per key/segment or most-asking node, in order to improve data placement decision
- Sequencer: serializes request for strict ordering
- Virtual synchrony: broadcasting events to other *Orchestrator* [15] for fault tolerance

Proxy *Proxies* render *Coordinators* accessible through the network. Because of the synchronous operation, *Translators* that keep locks can cause some serious burdens. For example, a sequencer will block all future client requests indefinitely, until the previous operation completes. As will explain in the next Section, our proxy design relies on full-duplex communication and heartbeat mechanism, between the clients and the *Coordinator*. For the proxy implementation, we used the High-Performance Remote Object Service Engine (Hprose) [53] framework and Websocket [33].

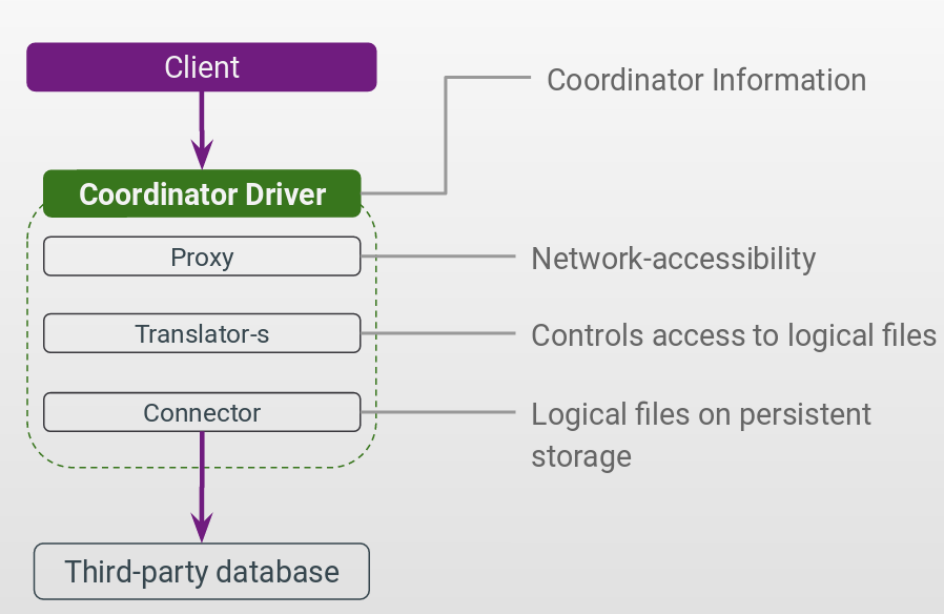

Figure 5.1 – Synthesis Service for Coordinator drivers. It piggybacks components from the repository into a synchronous stack. The Connectors implement shared logs on top of external databases, for recording transactions and metadata. Translators coordinate how the client access to the log, thus controlling the desired consistency level. Proxies make the *Coordinator* available to the clients through the network.

5.3 Driver Runtime

A transaction is a logical unit that is independently executed for data retrieval or updates. In order to achieve atomic completion of a distributed transaction, researches introduced a technique known as two-phase commit (2PC); an atomic commitment protocol that ensures that a

```
# Key range for which the Coordinator is responsible
"A-K":
    # Database client
    Connector:
      plugin: "boltdb"
    # Request-processing layers
    Translators:
      "0":
        plugin: "sequencer"
        blockw2r: true
      "1":
        plugin: "visualizer"
        output : "/tmp/requests_KeyRange"
    # Make Coordinator accessible through the network
    Proxy:
        plugin: "hprose"
        uri: "10.200.0.2:8880"
```

Listing 5.2 – Composition of customized Coordinator driver in the Keyzone language. The identifiers represent the range of keys for which this coordinator is responsible. The translators are stacked according to their numerical order.

transaction commit is implementing in the situation where a commit operation must break into two separate parts [107].

Next, we discuss two types of provided transactions. The *Update* and the *View*. The first is about allowing a client to write some changes of a top-level entity (e.g., logical file), and the second is about reconstructing the data of a top-level entity, on reading. In both cases, the client must open the corresponding type of transaction from the *Coordinator* and transfer data to, or from the *Devices*. The *Coordinator* does not participate in the data transfer.

5.3.1 Update Transaction

The *Coordinators* are meant to perform access control of logical files when multiple clients wants to access them. To do so, when a client wants to write() to a logical file it must contact the *Coordinator* and ask to open an *Update* transaction. If successful, the client store data to collections on the *Devices*, named after the Transaction Identifier (TID). Figure 5.2 illustrates the process.

- Commit-request phase: the client asks the *Coordinator* to precommit information about the collections on the affected *Devices*, for the forthcoming operation. If commit is possible, *Coordinator* records the request to the intention ledger (Figure 5.3). Otherwise it directly rejects the request.
- Commit phase: at the end of data transfer, the client sends to the *Coordinator* a request with the metadata, asking to persist it (Figure 5.3). On commit(), the *Coordinator* appends

the metadata as a record in update log and atomically removes the record from the intention ledger. On abort() the *Coordinator* moves the entry from the intention log to a global garbage collection log. In case of a crash between the two phases, higher layers can query the ledger for the incomplete transaction, extract the affected *Devices*, and remove the partially written data.

5.3.2 Ordering

Two-phase commit only promises that a transaction is atomic. Outside the transaction, the perceived behavior by the clients depends on how locking works in the database. Aiming for high concurrency and versatility, the lower handlers in the driver stack (Connectors) implement multi-version concurrency control (MVCC) [13, 22]. MVCC allows view transactions to read the last committed values while update transactions are writing new data. The data of an update transaction is not visible to any other transaction until the transaction commits.

Because the update log contains the modification history of a logical file the insertion order is of utmost significance; a wrong order will cause the end-user to see corrupted data. Since most of the databases do not natively support append operations, we exploit the fact that they order keys in a byte-sorted manner. Every new record gets a monotonic and sequential key so that the returned records will appear in the correct insertion order.

Coordinators also support file leasing for exclusive write access. For leased files, only calls that include the current TID can modify the entity. Nevertheless, it depends on the sequencer, or other similar translators, whether concurrent requests for the same key will reach the connector or will block waiting for the previous request to finish.

5.3.3 Leases over locks

Coordinators operations are synchronous; they acknowledge a request only after the request is complete. Because *Coordinators* do not share state, there is no easy way for the clients to know if the system has queued the request or if the network ceased it. Resending the request after a timeout would potentially solve the problem, but it would also require significantly more complicated layers to avoid duplicates and achieve "at most once" guarantees. If the client for which the *Coordinator* holds the lock, crashes, the overall progress will block forever. To avoid such deadlock, we employ leases instead of locks. They are locks with cancellation context and are revocable either explicitly (e.g., success or graceful abort) or implicitly (e.g., timeout).

5.3.4 Heartbeat

For leases to work, there must a way to update them when a transfer takes longer than the leasing period. One way to do that is for the client to ask the *Coordinator* to update the lease explicitly. That, however, requires additional coding on the client and knowledge as to time

intervals. An alternative way is for the *Coordinator* to maintain a permanent connection with the client for the duration of the transaction. As long as the connection remains alive the *Coordinator* automatically renews the lease. If it senses a broken connection, it does not update and let the lease expire.

The same mechanism is also useful for simple locks; the heartbeat removes the lock immediately when it senses a broken connection. There is, however, a corner case. It works only for in-memory locks on a single host. If locks are persisted or held on another host (e.g., distributed transactions), then it takes more sophisticated protocols to decide the action after a crash. In general, the default action for locks is to recover since clients expect the operation to terminate gracefully. If they receive no reply the behavior is undetermined. Oppositely, the default action for leases is to cancel the request and proceed with the next.

We implement the above mechanism using Websockets [33]. It is a network protocol that enables interaction between a web client and a web server with lower overheads, facilitating real-time data transfer from and to the server. By providing full-duplex communication channels over a single TCP connection, it allows messages to be passed back and forth while keeping the connection open. It also standardizes a way for the server to send content to the client without being first requested by the client. At any point after the handshake, either the client or the server can choose to send a ping to the other party. When the ping is received, the recipient must send back a pong as soon as possible.

```
1  func (s *Sequencer) BeginWrite(key string, tid string,
2      payload []byte) error {
3      // Semaphore like - add tid to the locks for a key
4      s.writes.Lock(key, tid)
5
6      // Step 1: forward begin request to the next layer
7      err := s.Stackable.BeginWrite(key, tid, payload)
8      if err != nil {
9          s.writes.Unlock(key, tid)
10         return err
11     }
12
13     // Step 2: the client can start I/O to devices
14     return nil
15 }
```

Listing 5.3 – Commit-request phase of the two-phase commit protocol. A client sends a request to pre-commit information about the affected devices (payload) of a transaction (TID). The receiving handler does its pre-processing and invokes the lower handler. Recursively, the request reaches the connector which adds the request as a record to intention ledger

```
func (s *Sequencer) EndWrite(key string, tid string,
    payload []byte) error {
    // Step 3: forward begin request to the next layer
    err := s.Stackable.EditEnd(key, tid, payload)
    if err != nil {
        return err
    }

    // Step 4: post-process (unlock the key)
    s.writes.Unlock(key, tid)

    // Step 5: Atomic transaction complete
    return nil
}
```

Listing 5.4 – Commit phase of the two-phase commit protocol. A a client sends a request to commit the metadata (payload) of the transaction (TID). The receiving handles invoke the lower handler before taking any action. Recursively the request reaches the connection which atomically removes the TID from the intention ledger and adds a new TID record to the update ledger. Once the operation is successful, the handlers can proceed with the post-processing.

5.3.5 View Transactions

Essentially, every *Update* transaction comprise a new version of the logical file. Doing so, enable developers to access particular changes in the logical file as to how they access particular versions on a revision control system (Table 5.1).

When a client wants to read() a segment of the logical file, it must contact the *Coordinator* and ask to open a *View* transaction. In turn, it returns a copy of the non-masked records of the update ledger (see next paragraph). In order to exclude these " versions" from the garbage collection process while the client is still reading we assign them a lease. We regard this type of *View* transactions as "protected". Being "protected" does not mean being "locked". Everybody, apart from the garbage collection, can still access the records. *Coordinators* also support an "unprotected" *View* operation. It returns the records in a single step without any leasing, and without needing the client to explicit close the transaction. The risk is that the garbage collector or any other system-induced may hit and compromise the data. Its useful for informative purposes, such as calculating the file size.

5.3.6 Update Record Masking

A notable difference between files and objects is that objects are immutable (write-once-read-many) whereas files are mutable (write-many-read-many). The last written data for an object is what will return on the next read(). Files are quite different. A client may open a file, write to an offset, and close it; then open it again, write to another offset, and close it. Thereby, reconstructing file data may involve multiple records. The exact reconstruction process will be

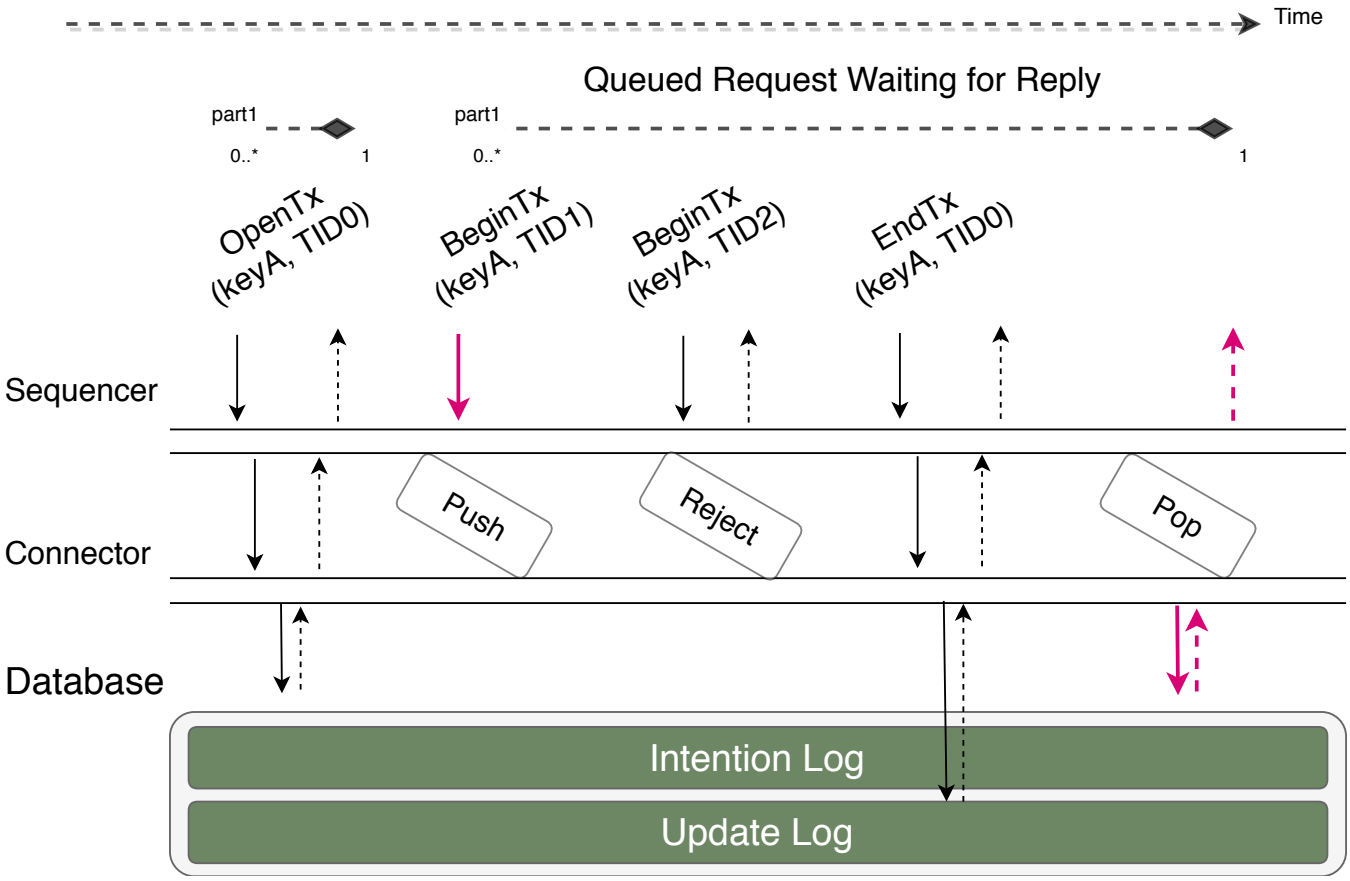

Figure 5.2 – Operation of the synchronous stack of the Coordinators. When a client starts a transaction the intentions (affected devices) are persistently stored in the intention log. The coordinator does not interfere in the critical I/O (data)path, which is between the client and the devices. When the client closes a transaction, the Coordinator persistently stores the transaction metadata to the update log - and remove the record from the intention log, atomically. The demonstrate sequencer has a queue depth equal to 1. If there is a pending request, the queue rejects the rest. Because the stack is synchronous (backend must confirm the operation before a handler replies), the client with the queued request may wait for an arbitrarily long time (we support leases and heartbeats).

discussed in Chapter 6.

The problem is that a client does not how many records there are, and the *Coordinator* does not know the exact nature of the logical entity; so does not how many records to return. Returning everything would be a solution, but it would increase the communication overhead and would put more effort on the client to resolve which records to use. Such an approach is also prone to failure in a highly concurrent environment with many read()s, write()s, delete()s. To better understand it let us draw an example. A client has started a read transaction for a file and is reading some data. In the meantime, another client removes the file. One solution is to implement strict locking, but that would kill the performance. The other is to queue the remove request until the commit of the read transaction - and block the client for an arbitrarily long time. An alternative, asynchronous, solution is to flip a flip and mark the file as removed. Such a solution has corner cases like, for example, what happens if a client removes a file, but in the meantime, another client starts

Keyzone API	Return	Comments
Info(name string)	(Info, error)	Return key information
Create(Key, Reset)	error	Create a new or rim an existing log
Set(Key, Landmark)	error	Log-appended presentation instruction
BeginUpdate(Key, TID, Payload)	error	Append targets on intention ledger
EndUpdate(Key, TID, Payload)	error	Append version on update ledger
BeginView(Key, Filter)	([]Version, error)	Return versions for data reconstruction
EndView([]Version)	error	Remove versions locking (free to gc)

Table 5.1 – Coordinator API. Every key describes a logical file as a tuple of an intention log and update log. In order to start writing a logical file, the clients must acquire an Update Transaction. Essentially, every *Update* transaction comprise a new version of the logical file; as a version on revision control systems. BeginView() traverses backward the update log until it finds a landmark. The landmark is an instruction as to how to reply

writing that file.

Our proposed solution is what we call “landmarks”. They are special entries in the log used as instructions for the ledger as to how to reply to incoming requests. When a client asks for a read transaction for a key, the ledger iterates backward the update log and fill entries to the message. If it stumbles onto a landmark, the landmark dictates how to reply. Currently available landmarks are *IgnorePrevious* and *Disappear*. The former dictates the ledger to reply immediately without further iterating the log — records located after the landmark become candidates for garbage collection. The Disappear keyword make the key disappear. If met, the textitCoordinator will as if there is no entry for the key. However, data remain intact and are not garbage collected. If need the users can remove the landmark, and the data will become again available, e.g., to undo a remove operation.

Log Trimming Depending on the access pattern of the application, the version log can grow and impose delays on the requests. To avoid such case, a background process on the *Orchestrator* periodically scans the log and move out-of-date versions to a global garbage collection log. Obsolete versions are those preceded by a landmark and do not participate in an active read transaction.

5.4 Summary

In this Chapter we present a framework for composing *Coordinators* for namespace management, metadata management, and distributed synchronization. The *Keyzone* is a declarative language that abstracts *Drivers* as a stack (pipeline) of basic handlers. We provide handlers for various databases, for controlling the consistency-level, to visualize control requests, and monitors to inform upper layers about the access pattern of a logical file.

The *Coordinator* API is build around the concept of a shared logs for recording modification

on a logical file. An agreed upon global ordering becomes simply the order of events in the log. The upper layers in the stack control access to the log, thereby controlling the consistency level.

5.4.1 Related Work

Metadata Catalog Service (MCS) [115] provide a mechanism for storing and accessing various types of metadata and allows users to query for data items based on desired attributes. BookKeeper [8] is a replicated service to log streams of records reliably. In BookKeeper, servers are "bookies", log streams are "ledgers", and each unit of a log (aka record) is a "ledger entry". Perhaps the most advertised ledgers are those based on Blockchain [118] technology; a growing list of records, called blocks, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. By design, a Blockchain is resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently, verifiable, and permanently". Essentially, any application that requires appendable storage can replace their implementations with a ledger, that additionally has the advantage of scaling throughput with the number of servers. BookKeeper client writes metadata about the ledger to ZooKeeper [54]; a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It provides a broad set of API calls with variable semantics so that users can decide the balance between availability and consistency.

Malacology [111] introduced a novel way for exposing internal services and abstractions of Ceph as building blocks for third-party applications. That is for third-party applications to benefit from years of code-hardening and performance optimization efforts that were necessary for users to entrust their data to the storage system. The authors used existing internal abstractions to compose two new higher-level services: a load balancer for file system metadata and a high-performance distributed shared-log. Cudele [110], from the same authors, stepped on those abstractions to present a framework and API that lets administrators specify their consistency/durability requirements and dynamically assign them to subtrees in the same namespace. Doing so allows administrators to optimize subtrees over time and space for different workloads, on a per-application basis.

Chapter 6

Virtual Storage Infrastructure

Conventional data-store assume that data is the primary and permanent asset, and applications come and go. In this data-centric architecture, the data model precedes the implementation of any given application and will be around and valid long after it is gone. Taking that as granted, justifies applying one-size-fit-them-all policies to all the applications. Many people may think this is what happens now or what should happen. However, that is very rarely the case. Businesses want functionality, and they purchase or build application systems. Scientists wish to simulate physical phenomena, which may be computationally intensive or I/O intense. Each application system has its data model and inextricably tied code with it. In this Chapter, we introduce a framework for constructing application-tailored storage (ATS); a middleware that separates data-management logic from the business and computation logic of an application. Alternatively stated, it separates changes made to application codes by science users from changes made to I/O actions by developers or administrators. Defer data-management decisions until the deployment phase, which is the key for portability; least authority security; storage federation over non-collaborative storage vendors; second-order scaling independently to the data-stores; and policy-based data distribution with criteria such as resiliency, performance, storage efficiency, and cost. ATS is not meant to replace existing general purpose data-stores but provide a data management layer that insulates the application from the physical storage infrastructure; it leans toward "application is the asset and can use any combination of existing storage systems". When the application terminates, so does the ATS.

6.1 Synthesis

The middleware is a client-side library intended to be the foundation for other layers to run upon, with no dependence on any technologies specified above or below it. For example, it should not matter whether *Devices* feature burst buffers or not, neither on to how many *Devices* the data are sent.

Its design is adherent to the Lightweight File System's [92] philosophy; it is stateless and uses the primitives of remote *Services* for naming , in-transit processing and data distribution,

and data-persistence. On the lowest level, it federates underlying Microservices into higher-level grids, or meshes, that uses to build a decentralized virtual storage infrastructure. For the management of those grids, the library provides plugin interfaces so that developers can inject their custom data-management routines. Management aspects such as request concurrency, sessions, in-transit processing, placement decisions, data distribution, data layout, and reconstruction, are defined in a configuration file called *Manifest*.

The library also provides a set of user-friend persistent storage interfaces for developers to effortlessly integrate Tromos into their applications, without the need to learn yer another low-level API. However, integrating a library into an application resources intervention into the application's source code. For the cases where modifications are not possible (e.g., legacy applications or binaries), *Tromos* provides various gateways. For example, using the fuse gateway the administrators can mount the storage container as a normal filesystem. Hence, the application binary can seamleslly access the virtual storage infrastructure. Figure 6.1 depics the design of the middleware.

Residing entirely in userspace, it can leverage technologies such as RDMA, zero-copy, vectored I/O and non-volatile memories which have minimized access node-to-storage and node-to-node transfer times [80, 84, 111, 122]. More importantly, it can leverage existing userspace libraries and take advantage of cloud storage, which would be prohibitive in a kernel-based implementation.

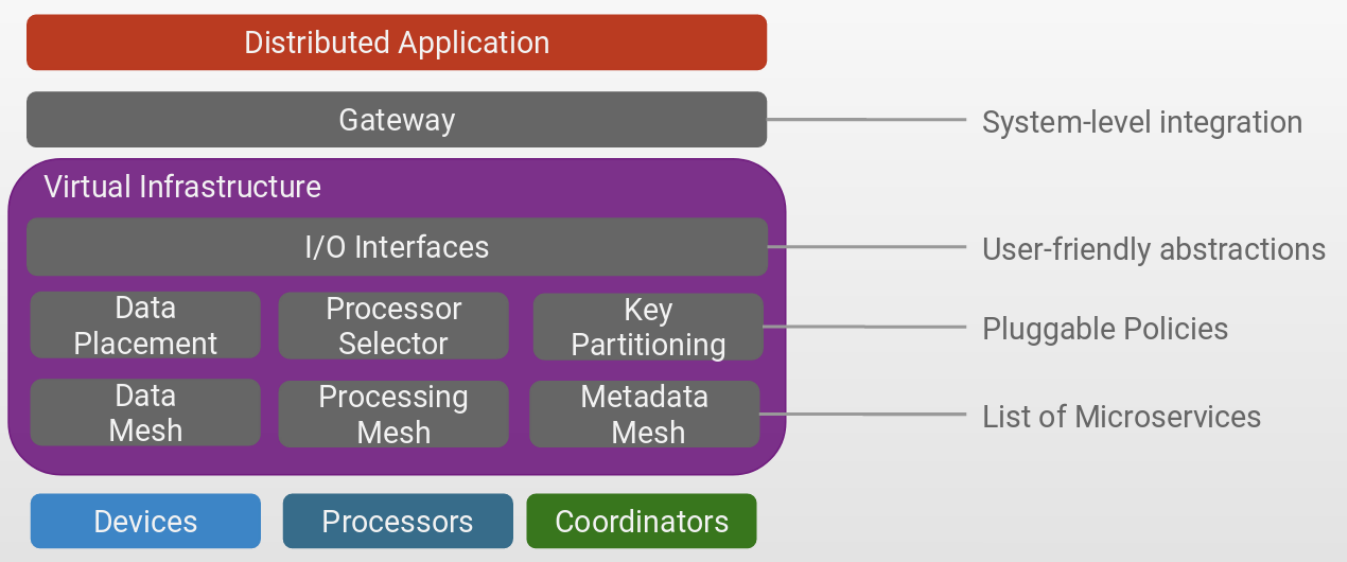

Figure 6.1 – Tromos client-side middleware provides access to the underlying virtual infrastructure. Application can use directly the library, or use the gateways for system level integration (e.g, use the fuse gateway to mount a storage container as a normal filesystem).

6.1.1 Manifest

The *Manifest* is a declarative language for describing the target virtual storage infrastructure in a definition file. By separating the management data logic from the application computation or business logic of the application, the application architects are able to tailor the storage environment to the requirements of the application at hand. For example, create a balanced namespace by injecting a Distributed Hash Table (DHT) plugin, or use prefix-based plugin to direct the request to a particular Coordinator with the desired Quality of Service (QoS) characteristics.

In the *Manifest* we identity three grid sections the information of which use the runtime to decide how and to which services to invoke. Each of those grids is individually scalable by bringing up and down *Service* instances. High scalability derives from the fact that there is little or no dependency on a centralized *Services*, therefore essentially removing this limiting bottleneck from the scalability equation. For example, Scalability on the data path is feasible by sharding data to *Devices*. Scalability on the control path is feasible by partitioning keys to *Coordinators*.

- **Messaging grid**: realizes a scalable global namespace by partitioning key range to distributed *Coordinator* instances. A unique feature is that the ability for differentiated key access. For example, keys prefixed with "unsafe_" string will get forwarded to a *Coordinator* that allows for high concurrency. Otherwise, requests go to serializable *Coordinators*. *Manifest* language makes trivial to describe such ad-hoc schemes, and *Tromos Deployer* automatically realize them without any intervention from the developer.
- **Data grid**: realizes tiered management of available *Devices*. Given a set of diverse and heterogeneous *Devices*, the data placement policies can adjust to reduce I/O completion time, maximize throughput, or data and load balancing for efficient *Device* utilization, i.e., align placement objective to the application requirements.
- **Processing grid**: realizes composite *Datapath* that span across multiple nodes. Such feature makes it possible to move data filtering logic from computational nodes to staging nodes closer to the data-store. Previously discussed gains such as portability, storage federation, and cross-platform scalability derive from this mechanism. For example, data may get transformed, routed to cloud providers, split across on-premise and cloud storage, and many other scenarios, without any changes into the application source code.

6.1.2 Composite Namespace

Clustering is crucial for *Coordinators*. A *Coordinator* cluster is a decentralized distributed system that provides a lookup service for (key, value) pairs. Spreading keys across a network of *Coordinators* circumvent single-node bottleneck, improve system resiliency against failures, increases the number of accomplished requests per second, and minimize queue latency per request. Sharing a common information source enables any participating node to retrieve the

value associated with a given key efficiently. *Tromos* realizes a scalable global namespace using *Coordinator* primitives.

A unique feature is that the ability for differentiated key access by partitioning namespace to *Coordinator* distinct characteristics. The idea relies on i) clients must contact a *Coordinator* and acquire a transaction before starting I/O ii) *Coordinators* are deterministically selected in a shortest-prefix manner iii) how the *Coordinator* will handle the request depend on pluggable stack drivers. For example, keys prefixed with "unsafe_" string will get forwarded to a non-serializable *Coordinator* that allows for high concurrency. Otherwise, requests go to serializable *Coordinators*. *Manifest* language makes trivial to describe such ad-hoc schemes, and *Tromos Deployer* automatically realize them without any intervention from the developer.

6.1.3 Device Management

Depending on the *Processor* driver, a client operation may involve more than one *Device*, e.g., stripping, mirroring, or erasure coding. Selecting the same *Device* twice within a single operation will lead to nondeterministic data interleaving and the user will retrieve corrupted data. A potential solution would involve a device manager that maintains a lookup with allocated devices per transaction, but that would make a single point of congestion. It would also put a constraint on the supported metrics. For example, under the circumstances, the plugins may augment the selection process with custom metrics like bandwidth and round-trip time, from the client to *Devices*. To this end, a centralized *Device* selection approach would not be viable since link-related information changes from client to client.

Instead, we propose a distributed *Device* manager. In *Tromos* approach the client-side engine also embeds a minimal *Device* management to arbitrate *Device* assignment. When clients initiate a new *Datapath* they include in the RSVP request a callback URI to the local device manager. When the *Datapath* leaves receive the RSVP they use the URI to contact the manager to claim a *Device*. Claims are uniquely identified using the RSVP identifier and the leaf identifier. Once a leaf acquires a device, it reports to the input port of the graph. When all the leaves have reported the root input port removes the lock from the device manager. The *Device* channels guarantee isolation between transactions so that two different transactions can safely write simultaneously to the same device. On the functional level at least, because in practice there will be interference. Future work is to include an SCSI reservation on the *Devices* to circumvent transaction interference.

6.1.4 Device Selection Strategies

Modern applications exhibit a variety of I/O patterns such batches that care for raw sequential throughput and interactive queries that care for lower-latency and random access. Continuous improvements in memory, storage, and network devices, both on-premise and in-cloud, introduce new challenges and opportunities in tiered storage management. Depending on the application

nature the data placement objective may aim to reduce I/O completion time, maximize throughput, or balance load and data across nodes, or any combination of the above [61]. Unfortunately, different objectives require a different set of metrics. For example, when the objective is the throughput, appropriate metrics include the type of storage device or the network bandwidth; when the objective is low latency, appropriate metrics include the current load on the storage and the network; and when the objective is load balancing the metric is the remaining capacity on the device.

Tromos abstracts the *Device* selection process in a well-defined interface so that storage administrators can integrate custom placement policies as plugins [127]. Researchers may also benefit from that abstraction as it makes it possible to experiment with novel placement algorithms without intervening into the system code. In order to support intelligent placement decisions, *Tromos* provides the plugins with a broad set of cross-layer information acquired both statically and at runtime that include both static and runtime information.

- Application-oriented hints (e.g., placement prefixes)
- Model hints (e.g., average filesize, maximum accepted delay)
- Advisory and mandatory hints from the *Processors* (e.g., compression, encryption)
- Access patterns from the coordinators (e.g., access frequency per host/global).

Below we present a list of supported *Device* selection policies.

Round Robin The purest form of algorithmic load balancing policies. It bases on circular iteration of the available resources. Albeit it does not account the properties or the load of the resources, its straight-forward implementation makes it ideal for choosing one out of many similar resources.

By Capability This policy consume *Device* information found in the *Manifest*. More specifically, *Device* annotation may describe the physical backend (e.g., HDDs, SSDs), quality of service (e.g., encryption, compression), or any other information such as geographical region. *Tromos* automatically process and index capability annotations. We reserve some keywords for special purposes. For example, $DISABLED$ excludes a *Device* from the selection process, either for maintenance or due to failures. When the developers or external provisioning tools edit the *Manifest*, the *Deployer* propagate changes to the peers. That is to ensure consistency between the *Manifest* and the system state.

By access pattern This policy consumes runtime information found in the *Coordinators*. Clients can ask *Coordinators* about the trends of a key (e.g., access frequency, most frequent client) in order to make intelligent data placement decisions. Such flexibility though comes with the cost of an additional request over the network which is not always needed.

6.2 Deployer

The framework *Deployer* treats the definition files as executable code for which it generates an execution plan describing what it will do to reach the described virtual storage infrastructure environment. Every time it executes a definition file it generates the same **idempotent and reproducible target environment**. Briefly, the deployment consists of three phases. In the first phase, it uses ssh to login to the remote peers and clones the central *Tromos* git repository that contains configurations, plugins, and graphs. In the second phase, it parses the *Manifest* to resolve the components for that host and retrieve dependencies using tools like yum and go deps. In the third phase, it composes the driver and bootstrap the service daemons.

6.2.1 Versatile topologies

ATS is not only about customized data-management, but also about the ability to setup virtual infrastructure in order to leverage the physical infrastructure to its full extent. Supported setups include:

Grid A grid is an alliance of loosely coupled machines with possibly very different hardware configurations which work together to solve a given problem/crunch data. The fundamental difference between a grid and a cluster is that in a grid each node is relatively independent and geographically distributed of others; problems are solved in a divide and conquer fashion. In a grid setup, there are decentralized islands of *Devices*, *Processors*, and *Coordinators* which remote clients can leverage.

I/O Forwarding In a tree data structure, the branches represent identical “thin” low-bandwidth link. In a “fat” tree, branches nearer the top of the hierarchy represent “fat” links with higher bandwidth than branches further down the hierarchy. For data-staging, we employ an inverse fat tree, with “thin” links between the compute nodes and staging or filtering nodes, and “fat” links from those I/O forwarding nodes to the storage. In such a setup, clients directly transfer their data to staging nodes that instances of *Processors* and *Devices*. The former removes the filtering process from the computational node and the second aggregates data and control access to parallel filesystems.

Peer-to-peer Peer-to-peer architectures partitions tasks or workloads among peers. Those peers connect directly, dynamically and non-hierarchically to as many other nodes as possible and cooperate to efficiently route data from/to clients. In such a setup, every host run dedicated instances of *Devices*, *Processors*, and *Coordinators*.

6.3 Storage API

The third, are high-level persistent storage interfaces that separate the user-visible interfaces from the virtual infrastructure [70]. Practically, they hide the raw API of the virtual infrastructure behind more user-friendly APIs such as files, object, or user-level libraries such as MPI-IO [120]. The need to support those interfaces drove the design of transactional primitives from *Coordinators*, reservation protocol from *Processors*, along with channels and collections from *Devices*.

6.3.1 Atomic Writers

Tromos wraps top-level operations into atomic Multi-resource transactions [89]. They are nested transactions that consist of a globally visible parent transaction acquired from the *Coordinator* and children sub-transactions local to the client. In general, a parent transaction associated with a collection on the *Device* and sub-transactions to data-transfer on that collection. For a parent transaction to commit, all the children sub-transaction must have committed first. If one fails, the whole transaction must fail and roll-back collectively. Changes written with a transaction become visible to other clients on after the transaction commit. Whether other transactions are running in parallel or not, depends on the *Coordinator*. Nevertheless, collection on the *Devices* guarantee isolation between concurrently running transactions. Replacing data inline, even if the underlying platform supports it, is not a viable solution since rollback would require significant state-keeping and maintain of previously written data on the memory of the host until the high-level transaction completes.

A handler is an in-memory structure that encapsulates end-to-end channels with the *Devices* and a managed transaction from the *Coordinator*. The latter also provides a key-value structure for storing the channel metadata and *Processor* state. A more detailed description of the handler functionality protocol, depicted in Figure 6.3, is as follows:

(i) The runtime initiates a new write handler with a unique transaction identifier number (TID) from a random number in 128 bit space [93].

(ii) The handler sends an RSVP request to the *Processor* to discover the *Datapath*

(iii) The leafs of the *Datapath* claim *Devices* from the *Device Manager*, using the *TID* and the leaf identifier.

(iv) The leaf replies back to the handler with the assigned *Device* identifier

(v) The handler claims a (parent) transaction from the *Coordinator*, including in the message the TID, the leaf identifiers and the identifiers of the assigned devices.

(vi) The handler creates channels to the selected *Devices*, named after the *TID* and assign *Datapath* leaves to those channels

(vii) Following I/O request write to streams on the end-to-end channel formed by the client, *Processor* and *Devices*

(viii) On closing, the handler triggers the *Processor* to flush the channel. Each stream represents an inner transaction, or delta.

(ix) When all inner transactions have committed, the handler commits the parent transaction back to the *Coordinator*

(x) The *Coordinator* appends the transaction to the update log

There are a few things to notice on the protocol. Handlers rely on a layered architecture that decouples metadata management from I/O operations, allowing clients to access in parallel access the file contents, and alleviate *Coordinator* from getting hammered by control messages. Such *close-to-open* semantics **removes a critical bottleneck** from the system and allows much higher performance without switching between throughput and metadata work. The parent transaction (update record) , represents an opened handler and inner transactions (deltas) represent writing operations within the handler (Table 6.1).

Second, it is the handler that decides the transaction identifier so to achieve **coherent naming** on the data plane (collections on devices) and control plane (transactions on coordinators). Third, **handlers are oblivious to exact drivers** running on the services which makes it possible to support various consistency, concurrency, and distribution models to apply without affecting the rest of the system. Next, we present a set of handlers that related I/O operations to channel operations differently, trading batch jobs for streaming and immediate consistency.

Append Handlers An append handlers drives all the I/O requests to the collection through the same *Processor* stream. The handler does not support update or delete operations given that a stream is a first-in-first-out (FIFO) structure optimized for append-only operations. There are no constraints as to the request length, but the handler must serialize request access to the stream to avoid interleaving data non deterministically. The primary usage of this “N requests, one stream, one delta” scheme is to transfer data across layers singularly. For example, due to the internal blocking functionality of the FUSE gateway (will be explained later), it translates user-level operations to multiple system-level operations of 128K bytes. From the handler perspective, these are independent operations and handle them as such. Append-only handler makes sure that data will be written in a contiguous chunk on the *Device*, thus making this handler ideal for usage in object storage.

Record Handlers

An alternative approach to the fixed-size block filesystems are the record-oriented filesystems; a class of filesystems that store files as a collection of records. Programs read and write full records, rather than bytes or arbitrary byte ranges, and can seek to a record boundary but

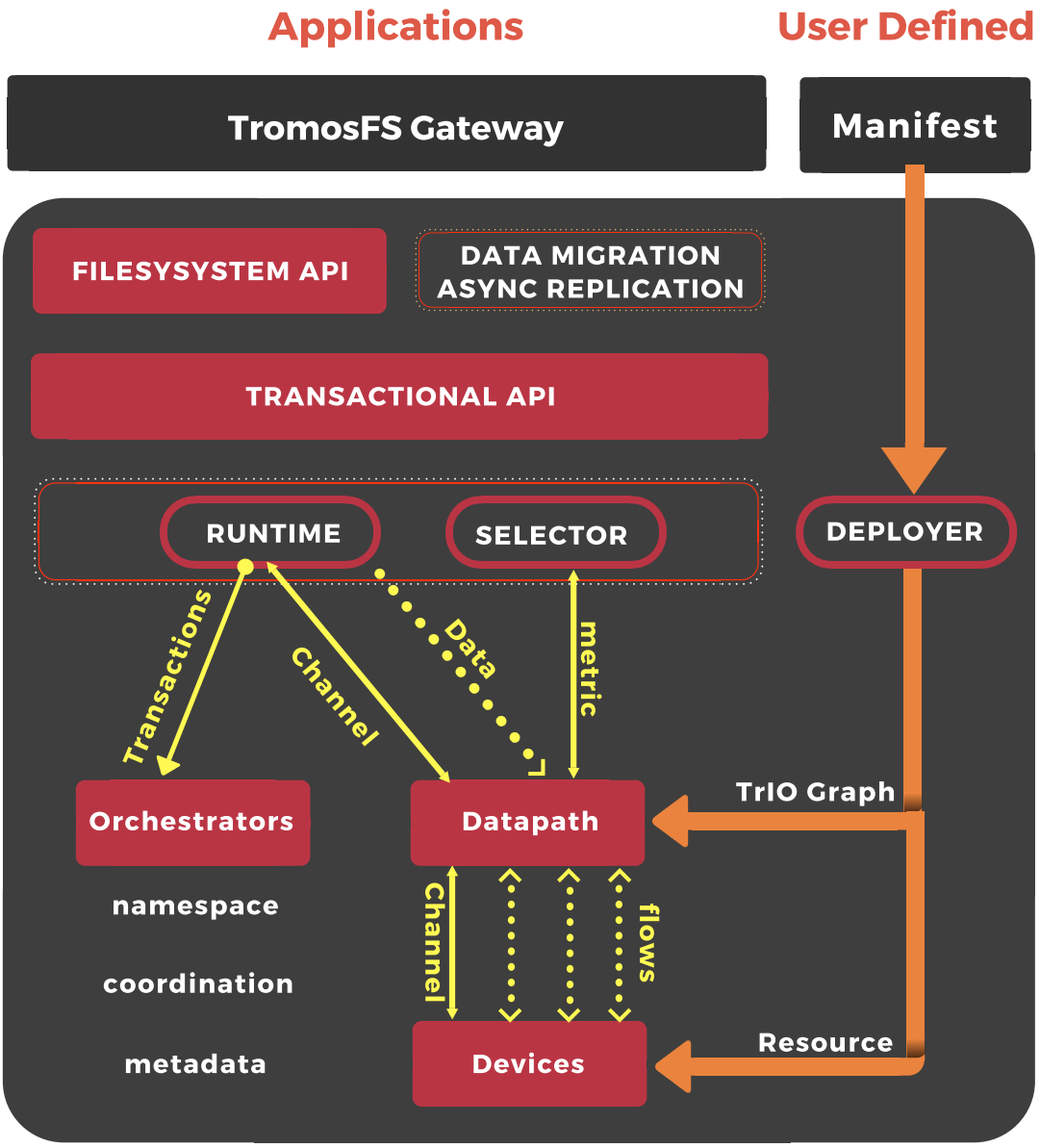

Figure 6.2 – Tromos Architecture: client-side engine use service primitives to build a virtual storage infrastructure, as described in the Manifest. The embeddable engine creates end-to-end I/O paths between the application and the *Tromos Devices*. For convenience, *Tromos* has built-in support for *Transaction API* and *filesystem API*. These APIs are used as primitives to build gateways

not within records. A record handler assigns incoming I/O request onto different streams, and thereby to different deltas. Their purpose is to solve the seekability deficit of append-only handlers. Each record is assigned a unique identifier that clients can later use to retrieve its content. Albeit there is no constraint of the record length, a fixed-length would render it ideal for usage as a virtual page; a fixed-length contiguous block of data, described by a single entry in the record table. When the handler receives a new write request, it spawns a new *Processor* and a new sub-transaction for holding request attributes and metadata. Streams terminate on chunks on the *Devices*, named after the record ID. On channel closing, the handler merges metadata

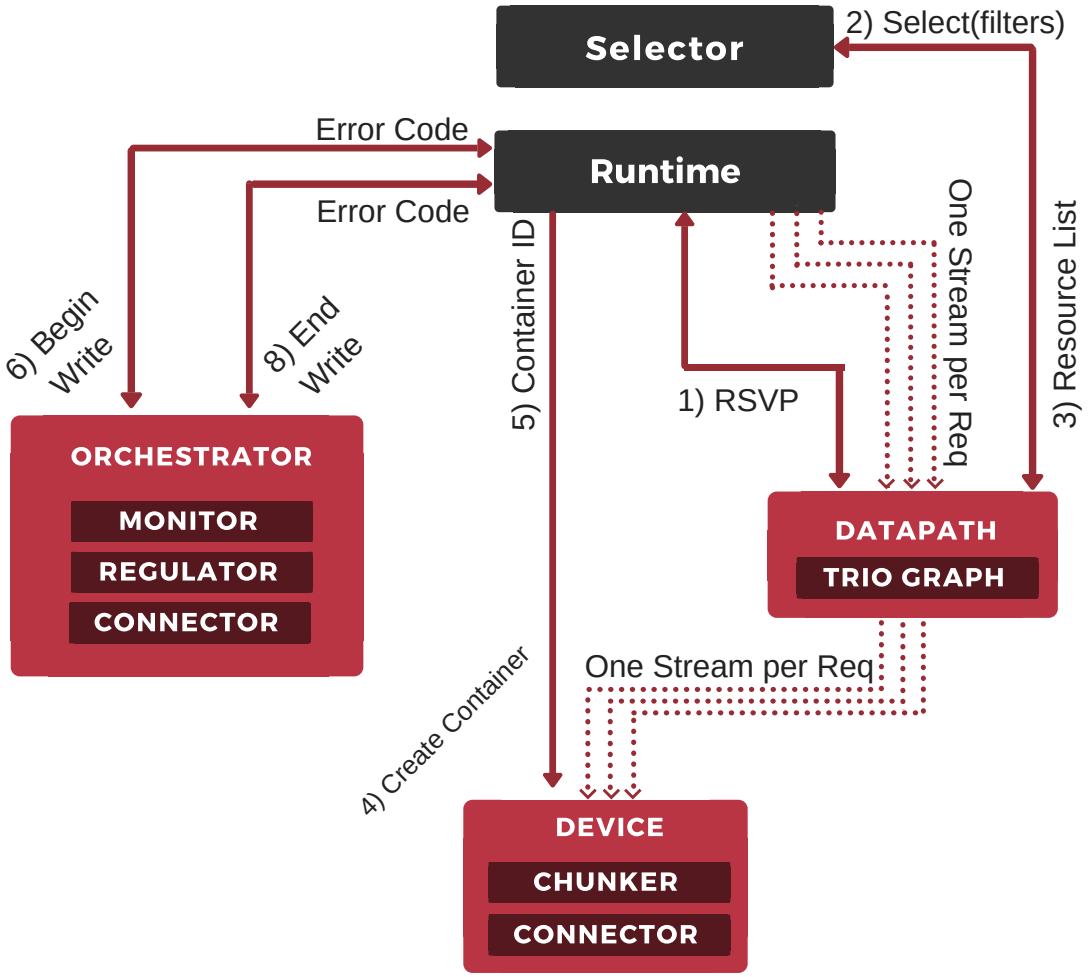

Figure 6.3 – Atomic multi-resource transaction protocol. The middleware creates end-to-end channels from clients to *Devices*, through *Processors*. Every different I/O request in the middleware cause the channel to spawn a new stream

to the respective sub-transaction, identified by the record ID. Such “N requests, N streams, 1 transaction” allows for random and parallel I/O, since records are indexed using the record ID.

Update Record	Type	Comment
TID	string	Transaction ID
GraphID	string	Used Graph
Sinks	map[string]string	Port-To-Device Map
DeltaList	[]Delta	Metadata of write operations
Delta Record	**Type**	**Comment**
Logical	format.Segment	Affected segment of logical file
Items	map[string]Item	Operation metadata
State	map[string]interface	Processor State

Table 6.1 – Handler information (Update record for the handler, deltas for every write request in the handler)

6.3.2 Readers

What clients usually perceives as a file is a logical file and not a physical file as perceived by the backend storage. A logical file is a view or representation of one or more physical files. For example, MPI-IO [120] based HPC applications split data to multiple chunks in order to circumvent the strong semantics of the parallel filesystem (e.g., Lustre [19], OrangeFS [94]). In *Tromos* deltas represent a unique write operation occurred to the logical files. Thereby, a logical file is a collection of deltas consolidated in a particular way. The consolidation method may change depending on the purpose and the semantics. For example, the objects in object storage are immutable. That means that a delta represents the original object content. Oppositely, files in the filesystem are mutable. Clients may open, write, write, close, open, write, close, or any other possible combinations. Thereby, we need to consolidate multiple deltas in order to represent the actual file content. *Tromos* abstracts the data reconstruction process as a plugin that exposes to the developers a list of deltas and a linear address space to place them. More specifically, the undertaken steps when the runtime opens a read handler for a key are:

(i) the handler initiates a read transaction to the *Coordinator* and retrieves the update records (snapshot isolation [22])

(ii) the handler flattens records to a list of deltas

(iii) the plugin filters the non-overlapping deltas using augmented trees [132] and places outcome in a linear address space that represents a contiguous memory space.

(iv) the handler allocates the represented contiguous memory and the trigger *Processors* to asynchronously fetch the data of filtered delta. (data-sieving [120]

Although the data will serve subsequent read requests for the handler in memory, the initial data reconstruction imposes a non-negligible overhead. To mitigate that overhead the *Runtime* provide Correctable [45] methods for speculative access to the partially filled memory buffer, i.e., preview data while they are still loading on the background.

6.4 Gateways

For applications to leverage *Tromos* they must embed the middleware engine into the source code. That may not always be possible due to binary-only applications, legacy code, or risks of jeopardizing a solid application. The last component of the framework are the gateways — standalone processes built on top of the above interfaces and make it possible to integrate the middleware on the system-level. For demonstration, we implemented a Fuse gateway on top of the SDK, in less than 400 lines of Golang code. It is stateless and translates POSIX operations to the respective calls of *Tromos* interfaces. Using that, the clients can mount and access the middleware like any other file system, without any modification into their source code. Other

gateways that are still under development include a driver for an FTP server, a web service for clients to select the *Manifest* to use, and a gateway for indexing data from external storage systems into the middleware; essentially importing data without copying them.

6.4.1 Relevant work

Software-defined storage (SDS) is a marketing term for computer data storage software for policy-based provisioning and management of data storage independent of the underlying hardware. In this umbrella term fall multiple different types of flow-processing and data-management tools. For example, DAOS [69] is a data-management layer running atop Lustre parallel filesystems. Its goal is to efficiently handle the physical infrastructure in a manner that improves utilization and quality of services. Clarrise [55] is a data-management engine that improves GPFS utilization by collecting information from all the layers participating in the I/O stack. Similarly, IOStack [44] is an engine that allows the injection and execution of code for in-transit processing before data get persisted in Swift object storage. The term SDS allows for self-managed data stores such as GlusterFS [100] and Ceph [130].

OpenSDS [66] introduced a cross-platform management tool for providing per-application policies for background tasks, such as scrubbing, replication, lifecycle, tiering and migration. Also, iRODS [97] is serving an increasingly important role in data management; it enables data discovery using a metadata catalog that describes every file, every directory, and every storage resource. In general, both tools focus on offline data processing while our proposition is primarily for use at transfer time.

Perhaps the most relevant work is Maxta [78]. It introduced a hyper-converged infrastructure for users to define storage attributes independently for each application.

```
Usage:
  tromos [command]

Available Commands:
  gateway     gateway handler
  help        Help about any command
  scan        Index the contents of existing systems

Flags:
  -h, --help   help for tromos

Use "tromos [command] --help" for more information about a command.
```

(a) Tromos console

```
Usage:
  tromos gateway fuse [command]

Available Commands:
  start       Initiate a fuse filesystem
  stop        Terminate a fuse filesystem

Flags:
  -h, --help                help for fuse
      --mountpoint string   Where fuse will be mounted

Use "tromos gateway fuse [command] --help" for more information about a command.
```

(b) Every step gives hints as to what arguments are expects

```
>> ./bin/tromoscli gateway fuse start --manifest ./evaluation/scripts/tromos-schemas/laptop.yml --mountpoint /tmp/test-mountpoint/
INFO[0000] Clean start for :gzone                        file="tromos/plugins/database/boltdb/boltdb.go:42"
INFO[0000] Store database to:/home/phobos/GO/src/tromos/bin/databases//gzone  file="tromos/plugins/database/boltdb/boltdb.go:53"
INFO[0000] Viewspace.Add :ID:gzone, Stackable:consistency <- monitor <- boltdb, Persistable:boltdb  file="tromos/swarm/namespace.go:36"
INFO[0000] ResourcePool.Add :ID:r0, Stackable:blob <- throttle <- filesystem, Persistable:filesystem  file="tromos/swarm/resourcepool.go:33"
INFO[0000] ResourcePool.Add :ID:r1, Stackable:blob <- throttle <- filesystem, Persistable:filesystem  file="tromos/swarm/resourcepool.go:33"
INFO[0000] ResourcePool.Add :ID:r2, Stackable:blob <- throttle <- filesystem, Persistable:filesystem  file="tromos/swarm/resourcepool.go:33"
INFO[0000] ResourcePool.Add :ID:r3, Stackable:blob <- throttle <- filesystem, Persistable:filesystem  file="tromos/swarm/resourcepool.go:33"
INFO[0000] Add to index:default seq:0 resource:r0        file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:scannable seq:0 resource:r0      file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:cleanstart seq:0 resource:r0     file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:default seq:1 resource:r1        file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:scannable seq:1 resource:r1      file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:cleanstart seq:1 resource:r1     file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:default seq:2 resource:r2        file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:scannable seq:2 resource:r2      file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:cleanstart seq:2 resource:r2     file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:default seq:3 resource:r3        file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:scannable seq:3 resource:r3      file="tromos/plugins/selector/byqos/byqos.go:42"
INFO[0000] Add to index:cleanstart seq:3 resource:r3     file="tromos/plugins/selector/byqos/byqos.go:42"
Mount triofs at: /tmp/test-mountpoint/
```

(c) Users can mount the customized data-store as any other filesystem.

Figure 6.4 – Tromos terminal

Chapter 7

Evaluation

In this Chapter, we evaluate and investigate the capabilities of *Tromos SDK*. To do so we define two main objectives. The first, qualitative, objective is to prove that *Tromos* is a versatile SDK that makes possible to implement at least the same functionality as to an existing system, without systems programming. The second is to provide quantitative performance measurements and provide insights about its pros and cons as compared to available storage solutions. More specifically, our goal is to show that by selecting proper components, the application can experience greater performance over running on top of a general-purpose storage system. Putting it simply, if the application does not require strong semantics, do apply such policy. If the application requires strong semantics, add them as a plugin, and so do for any other storage aspect.

Instead of merely laying commented results, in the next sections we discuss the required adjustments so to move from one experimental setting to another. In many cases, we do not directly present the final setting, but we go through the intermediate settings and discuss why they fail. The purpose to provide a guide-through for the reader to comprehend how individual *Tromos* components can affect the overall system's behavior.

7.1 Methodology, Tools And Setup

In order to investigate *Tromos* programmability we set compare it against RedHatś Gluster [100]. It is a scalable network filesystem suitable for data-intensive tasks such as cloud storage and media streaming. The reason for picking Gluster as the control platform is due to its native support for various volume types, which among others include throughput, resiliency, scalability, and storage efficiency. Although comparison with other systems would potentially be more sensible for certain scenarios, either they lack the programmability tenets of Gluster or would render the comparison less objective. For example, Lustre is a well-established general-purpose HPC storage used for parallel access. However, given that its implementation is kernel-based it would not be objective to compare it *Tromos* which is written entirely in userspace. Perhaps Ceph could be used instead of or along with Gluster since they are similar systems, but Gluster was preferred due to its programmability merits.

For the performance comparison, we evaluate *Tromos* and *Gluster* against a variety of workloads (synchronous, asynchronous), access-patterns (sequential, random), and operations (write, read). To produce all the potential combinations of workloads, access-patterns and operations we use fio [10] stress tester. More specifically, for the synchronous workloads we use the provided "sync"engine and for asynchronous the "libaio" engine. The former issues IO requests one after another, waiting for the first to complete before starting the next one. The latter enables threads to overlap I/O operations with other processing, by providing an interface for submitting one or more I/O requests in one system call without waiting for completion. We use it to show the implicit effect one request can have to its following, e.g., due to caching. For every combination we take into account three factors; the number of concurrently opened files, the iodepth, and the transfer size. The iodepth defines the number of parallel IO operations the application can issue to the OS against a file. For example, a sequential job with iodepth=2 will submit two sequential IO requests at a time.

For scalability comparison we evaluate *Tromos* and *Gluster* against parallel workloads produced by IOR (Interleaved or Random). It is a commonly used file system benchmarking application particularly well-suited for evaluating the performance of parallel file systems. Fio is a single-host stress tester and as such it cannot yield scalability insights. In this set of experiments, instead of using as factors the opened files and the files, we use the number of parallel tasks and various workload classes that represent real HPC application.

The allocated storage cluster for the experiments consists of 10 KVM-based virtual machines with each machine equipped with 4 CPU cores, 16 GB of RAM, CentOS 7.2.1511, Golang 1.9, and *GlusterFS* 3.7.8. Additionally, every machine had mounted four block devices of 20GB, formatted with XFS, that served as the backend for the Tromos devices and *GlusterFS* bricks. GlusterFS was accessible to fio via Fuse 2.9.2-6 and *TromosFS* via Golang Fuse. To wipe out any interference from other tenants in the virtual infrastructure all the experiments were conducted three times, on different dates. Due to relatively small variance among the acquired values (less than 8%) we present the average value of the three measurements.

7.1.1 Architectures

In Gluster, a volume is a logical collection of bricks where each brick is an export directory on a storage server in a trusted network. Each volume implements a unique way of managing the bricks, so to provide different functionalities on the higher level. In *Tromos* terminology, the bricks are *Devices* and the volumes are *Processors*. The primary difference between the two systems is that bricks and volumes are "hardcoded" in Gluster whereas the behavior of *Devices* and *Processors* in *Tromos* customizable. Figure 7.1 illustrates the process of creating a customized storage system. *Tromos* deployer uses the *Manifest* documentation of the target environment (left) to automatically produce and deploy the running environment to the proper host (right).

Gluster relies on extended attributes on the backend files to keep track of the operations. In general, its transactional model adheres to the following steps

i locks the logical file

ii set a dirty xattr* on the file on the backend

iii write to the file on the backend

iv clear the dirty xattr* and set pending xattrs* for failed writes.

v unlock the logical file

The dependence on attributes proved to be a limitation when fabricating the experiments. The initial goal was to run the brick over directories on a parallel filesystem. Gluster, however, was not able to use the filesystem attributes, so we had to present the files of the parallel filesystem as block devices to the virtual machines, and then format them as xfs filesystems. The directories of that filesystem comprise the Gluster bricks. In general, for the same functionality *Tromos* use the *Coordinator* and not assume as to the capabilities or properties of the backend, so it could seamlessly run atop the parallel filesystem directly. Aiming at an objective comparison, whenever possible, we define *Tromos* values to the default values of Gluster (e.g., block size, transfer size, parity blocks).

A notable difference between the two systems is the way they use consistent hashing [62] to implement a Distributed Hash Table (DHT). Gluster combines key management with data management, and therefore DHT participates in the data distribution logic. Each subvolume (brick) is assigned a range within a 32-bit hash space, covering the entire range with no holes or overlaps. When users open() a file, GlusterFS run a hash function that converts the file-name into a number. That function is about routing, not splitting or copying. Exactly one brick will have an assigned range including the file's hash value, and so the file "should" be on that brick. In this algorithmic approach, the physical location where clients should write data is predetermined. That aids on reading as clients can directly retrieve the content. Its drawback is that can potentially lead to "hot spot" and data imbalances, with some nodes serving significantly more data than others. Tromos, on the other hand, separates the key management from the data management. Targeting on intelligent data placement, clients in *Tromos* ask a decentralized device manager for the *Device* to write their data. The policy for device selection is pluggable. In our setup, the device manager chooses the next device in a round-robin manner so to ensure that all devices are equally loaded. Depending on the running state and conditions the *Device* may change from call to call.

Hence, the physical locations are not deterministic requiring the clients to query the respective *Coordinators* to get the metadata. In order to avoid congestion on Coordinators *Tromos* gives the possibility to distribute keys across multiple *Coordinators*. Thereby, the namespace is a composition of multiple *Coordinators*. The key partition policy is pluggable so that various topologies can be supported. To be compliant with Gluster, in our experiments we use consistent hashing to decide the *Coordinator* responsible for the key. This is however only a mere case. Other available policies include longest-prefix. That allows filenames with a specific prefix (e.g.,

small) to be served by a different coordinator than filenames prefixed with “large”. Given that *Coordinators* are independent, heterogeneous, and potentially geographically distributed, such an approach makes it possible to assign different priorities to different filenames.

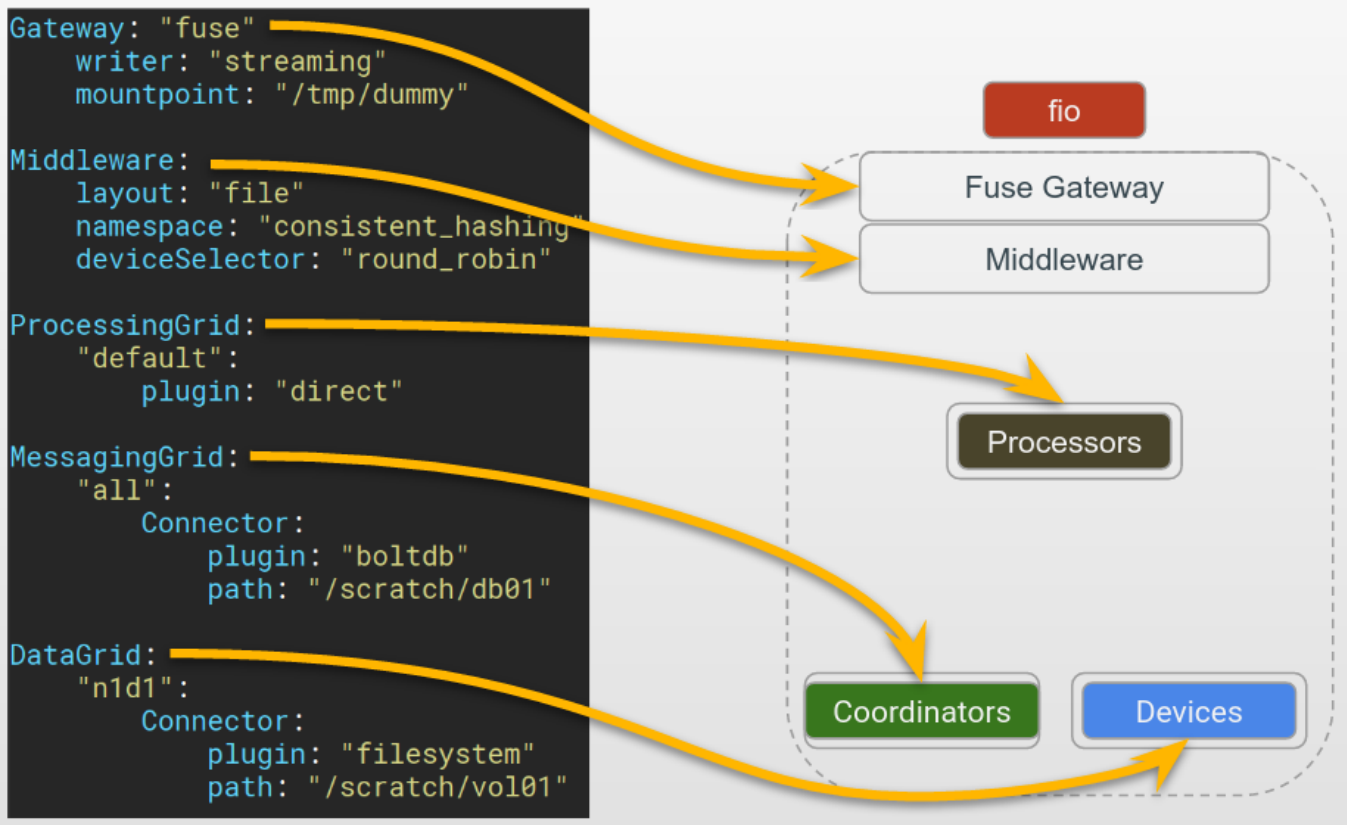

Figure 7.1 – [Storage-As-Code] Storage-As-Code: The developers model the target environment and *Tromos* automatically deploys it

7.1.2 Volumes

Below, we present a set of volumes for which we explain what it does, why we chose it, how GlusterFS implements them, how we realized them using *Tromos SDK*.

Distributed

Files in this volume are distributed across various bricks. This type of volume is used when the requirement is to scale storage and the redundancy is either not important or is provided by other hardware and software layers. So file1 may be stored only in brick1 or brick2 but not on both. The purpose for such a storage volume is to easily and cheaply scale the volume size. However, this also means that a brick failure will lead to complete loss of data and one must rely on the underlying hardware for data loss protection. The evaluation objective of this volume is to stress the differences on the control path, and more specifically the difference between algorithmically finding the data location versus the lookup query.

Striped Volume

Striped volumes are composed of equally sized stripes of data written across bricks in the volume. Such a volume divides a large file into smaller chunks (equal to the number of bricks in the volume) and stores each chunk in a brick. Consider a large file being stored in a brick which is frequently accessed by many clients at the same time. This will cause too much load on a single brick and would reduce the performance. With striped volumes, the load is distributed so that files can be fetched faster. For best results, striped volumes should be used only in high concurrency environments accessing huge files. The evaluation objective of this volume is to see the datapath behavior with lightweight CPU processing. Because the stripes are dependent and interleaved, a read operation must spend some time reconstructing the data. In *Tromos* we implement the equivalent of striped volume as a graph with a striping kernel. The number of outputs that correspond to stripes is given as a user argument to a graph macro.

Replicated Volume

Replicated volumes maintain exact copies of the data across on all the bricks in the volume. The number of replicas in the volume can be decided by the client while creating the volume. So we need to have at least two bricks to create a volume with two replicas or a minimum of three bricks to create a volume of 3 replicas. One significant advantage of such a volume is that data are still accessible even if one of the bricks fail. Such a volume is used in environments where high-availability and high-reliability are critical. The evaluation objective of this volume is to see the datapath behavior for memory intensive operations. It is intensive because it must locally buffer the incoming data before forwarding them to the replicas. In *Tromos* we implement the equivalent of replicated volume as a graph with a mirroring kernel. The user can define the number of outputs, that correspond to replicas, as an argument to a runtime macro in the graph.

Dispersed Volume

Dispersed volumes provide space-efficient protection against disk or server failures. It stores an encoded fragment of the original file to each brick in a way that only a subset of the fragments is needed to recover the original file. The administrator configures the number of bricks that can be missing without losing access to data on volume creation time. The evaluation objective of this volume is to see the datapath behavior with heavy CPU processing. Erasure coding also has the nice property that the upstream is computationally intensive whereas the downstream is computational free. That is, on writing it must compute the parity streams, which are only needed if an error occurs. Otherwise, the downstream of a dispersed volume is similar to that of a striped volume. In *Tromos* we implement the equivalent of dispersed volume as a graph with a Reed-Solomon kernel. For consistency we adhere to the default schema (8,2) used in Gluster, i.e., up to two devices can fail without compromising data integrity.

7.2 Synchronous Workload

In this first set of experiments, we evaluate the two systems against a write-once-read-many access pattern. More specifically, such a pattern involves large files written synchronously and sequentially. The most common applications of this category include media applications and archival application. In the context of cloud computing, a platform that exhibits the above properties is used as the backend for object-storage.

Code 7.1 shows the description of a streaming setting in which all application requests, as received by the fuse gateway, are driven through a single pipeline. Grids contain various Coordinator and Device services. The namespace plugin mandates the key partition to Coordinators whereas device selector plugins mandate how to distributed data to the devices. Hence, it is easily expandable to include an arbitrary number of coordinators and devices. On this experiment and given that metadata traffic is relatively low, we employ a single Coordinator running on the client host whereas devices are located on remote hosts acting as storage nodes. The reason is to demonstrate that services may run either locally or remotely.

```
Gateway:
    plugin: fuse
    writer: streamer
    mountpoint: /tmp/test

Middleware:
    layout: "file"
    namespace: "consistent_hashing"
    deviceSelector: "round_robin"

ProcessingGrid:
    "default":
        plugin: "direct"

MessasingGrid:
    "H-range":
        Connector: "boltdb"
        path: "/scratch/db/db0"
    "L-range":
        Connector: "boltdb"
        path: "/scratch/db/db1"
DataGrid:
    "n1d1":
        Proxy:
            plugin: "grpc"
            uri: "10.200.0.1:7000"
        Connector:
            plugin: "filesystem"
            path: "/scratch/vol01"
    "n2d1":
        Proxy:
            plugin: "grpc"
            uri: "10.200.0.2:7000"
        Connector:
            plugin: "filesystem"
            path: "/scratch/vol01"
    ..............
```

Listing 7.1 – Skeleton of a Manifest description for a customized virtual storage infrastructure. The setting can be easily adjusted to distributed, stripped, replicated and dispersed volumes by changing the I/O processor arguments

7.2.1 Distributed volume

Figure 7.2 shows the measurements for read and write operations against a distributed volume. The first gained insight of this measurement is that Fuse enforces an aggregated hard limit of 450 MB/s, both in cases for Gluster and Gluster. The reason behind this limitation is that

Fuse involves an additional data transfer (copy) between the userspace implementation and the kernel. More specifically, the application sends its data to Fuse, which moves the data to the kernel, which pushes data back to the userspace implementation of Gluster and Tromos. In turn, the implementations issue another call to the kernel to write the data either to the underlying filesystem or to transfer them through the network. The second insight is that *Tromos* significantly outperforms Gluster on reading. The reason for that is the internal behavior of the streamer writer. As previously said, on write(), the streamer forwards all the application requests (for a handler) to devices through a single pipeline. Equivalently for read(), the streamer brings the whole data from the devices into memory - handles data as objects. Despite the initial transfer cost, it seems that the cost is amortized over time as the in-memory data directly serve future reads.

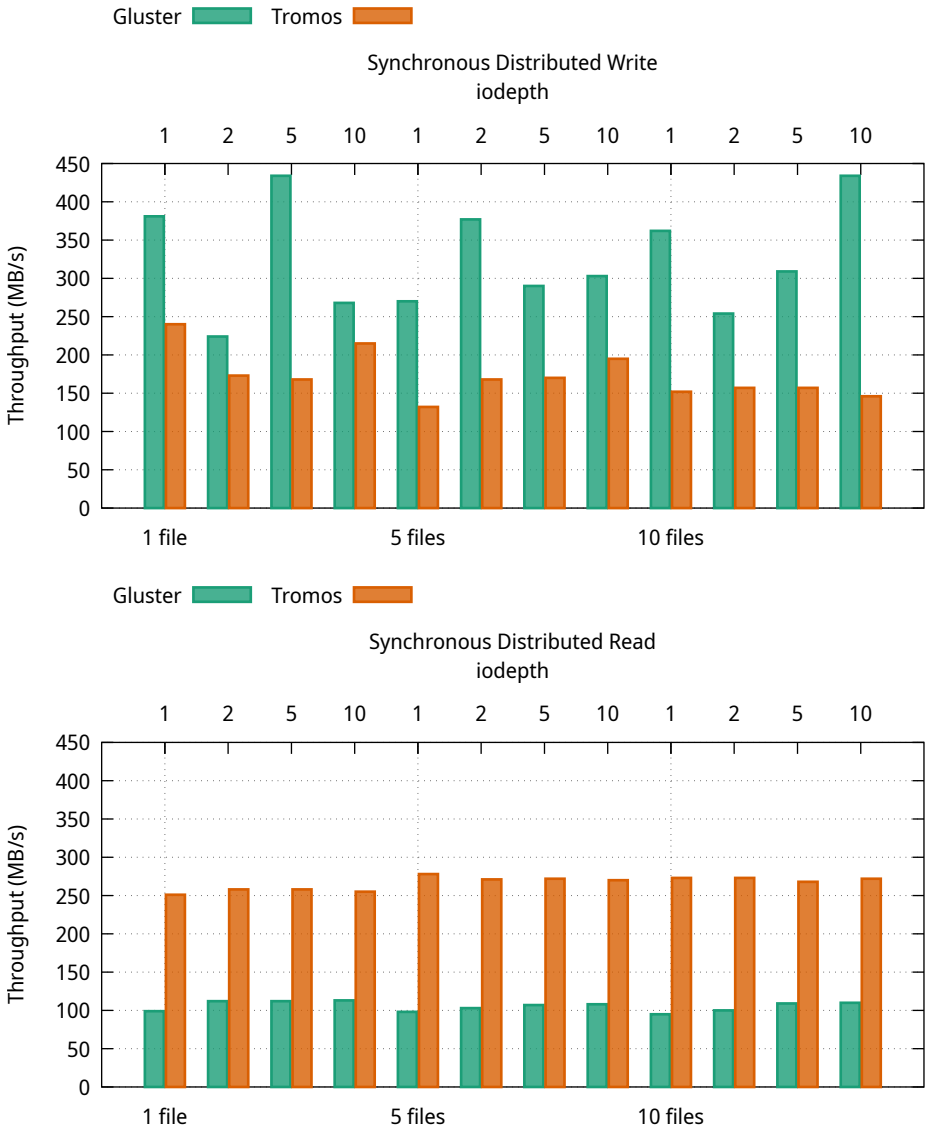

Figure 7.2 – Synchronous sequential I/O over distributed volume

7.2.2 Stripped volume

In order to support stripping, all that it takes for *Tromos* is to apply the changes of Code 7.2 to the skeleton of Code 7.1. Figure 7.3 shows the measurements for reading and writing operations against a distributed volume. There are two notable facts on these plots. The first is that after a certain point Gluster was causing a deadlock to the application. That was a reproducible behavior regardless of the workload type. A potential explanation may align with the comments of [68]

> ... One may have noticed already that so far we assumed a single location for storing data, e.g., local disk or remote node. The reason is that POSIX semantics were designed in such a way. POSIX compliance can be violated when stripping data across multiple server nodes. Imagine a scenario where a file's contents are 'AA' and the first byte is stored on server0 and the second on server1. Now one process overwrites with 'BB' and another overwrites with 'CC' while a third does a read. This is a race. Even on a local system this is a race but read() should return only one of three possible values: 'AA', 'BB', or 'CC.' In a distributed system, it is easy to imagine how this POSIX compliance can be violated and a read() might incorrectly return something like 'AB' or 'CA' or 'BC.' ...

The second fact is that for read, Gluster scales better than *Tromos* as the iodepth (the number of parallel threads) increase. That is an inherent deficit of any engine that is based on streaming. Given that a stream is an append-only structure, one request must block a previous request is writing on the stream. However, as the figure indicates, the number of concurrently opened files can amortize the serialization deficit.

```
1 ..............
2 ProcessingGrid:
3     "default":
4         plugin: "stripped"
5         stripes: 4
6 ..............
```

Listing 7.2 – Integration of Stripping processor, with 4 stripes

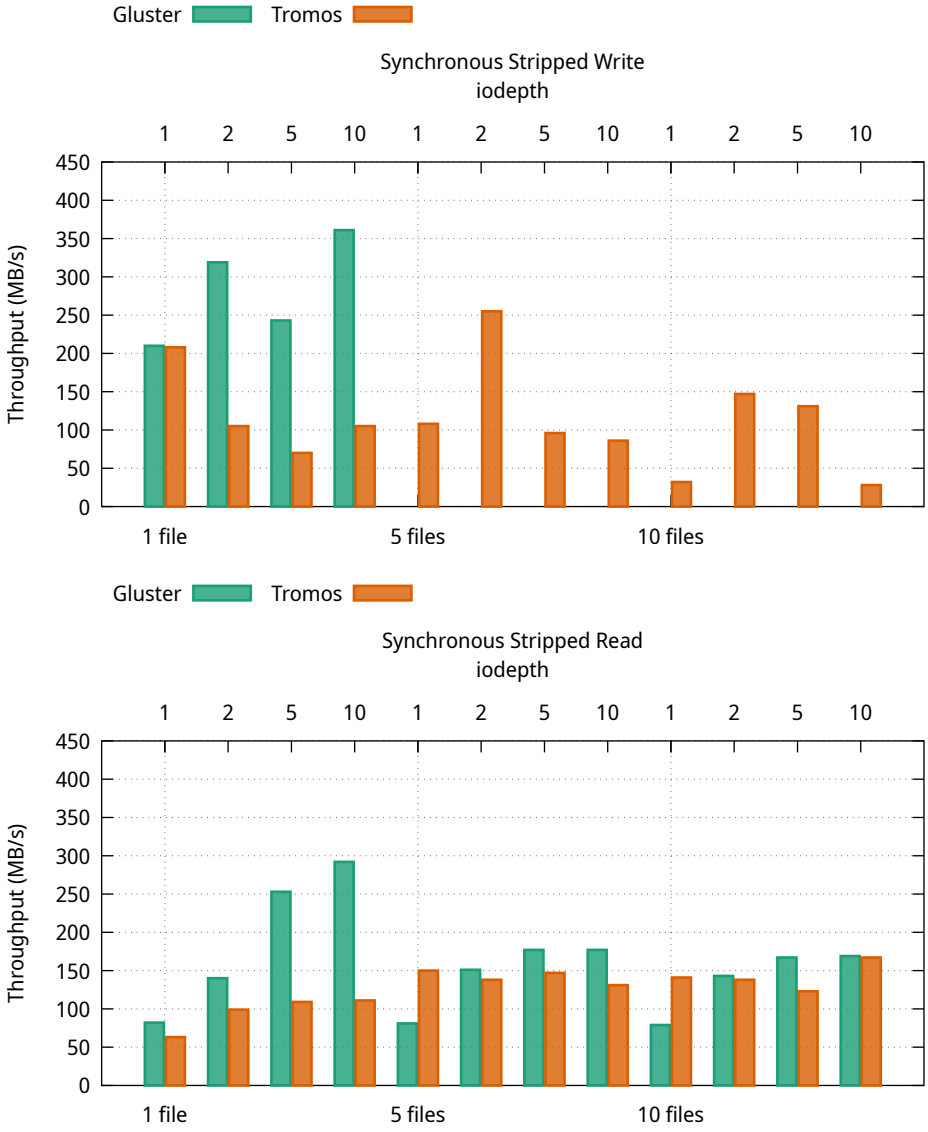

Figure 7.3 – Synchronous sequential I/O over stripped volume

7.2.3 Replicated volume

In order to support stripping, all that it takes for *Tromos* is to apply the changes of Code 7.3 to the skeleton of Code 7.1. Figure 7.4 shows the measurements for reading and write operations against a replicated volume. The results are quite interesting both in write() and read(). More specifically, in the case of a write operation(), the throughput of Gluster drops by a factor of 2.5 as compared to throughput of *Distributed* volume. That factor almost coincides with the number of replicas: three. At the same time, *Tromos* throughput remains relatively stable. That derives from the fact that *Tromos* datapath is highly concurrent and lockless. It is only bounded by the CPU-power and the available network bandwidth. Given that CPU-consumption is kept low and the bandwidth is much higher than the data-production rate, *Tromos* can keep multiple replicas without compromising the performance.

For read(), the throughput is the exactly same as in a *Distributed* volume, both for Gluster and *Tromos*. The reason is both *Tromos* and *Gluster* read data from a single replica - regardless of the number of replicas. A potential improvement would be to fetch different chunks from different replicas so to improve the reading rate.

```
1 ..............
2 ProcessingGrid:
3     "default":
4         plugin: "mirror"
5         stripes: 4
6 ..............
```

Listing 7.3 – Integration of Mirroring processor, with 3 replicated locations

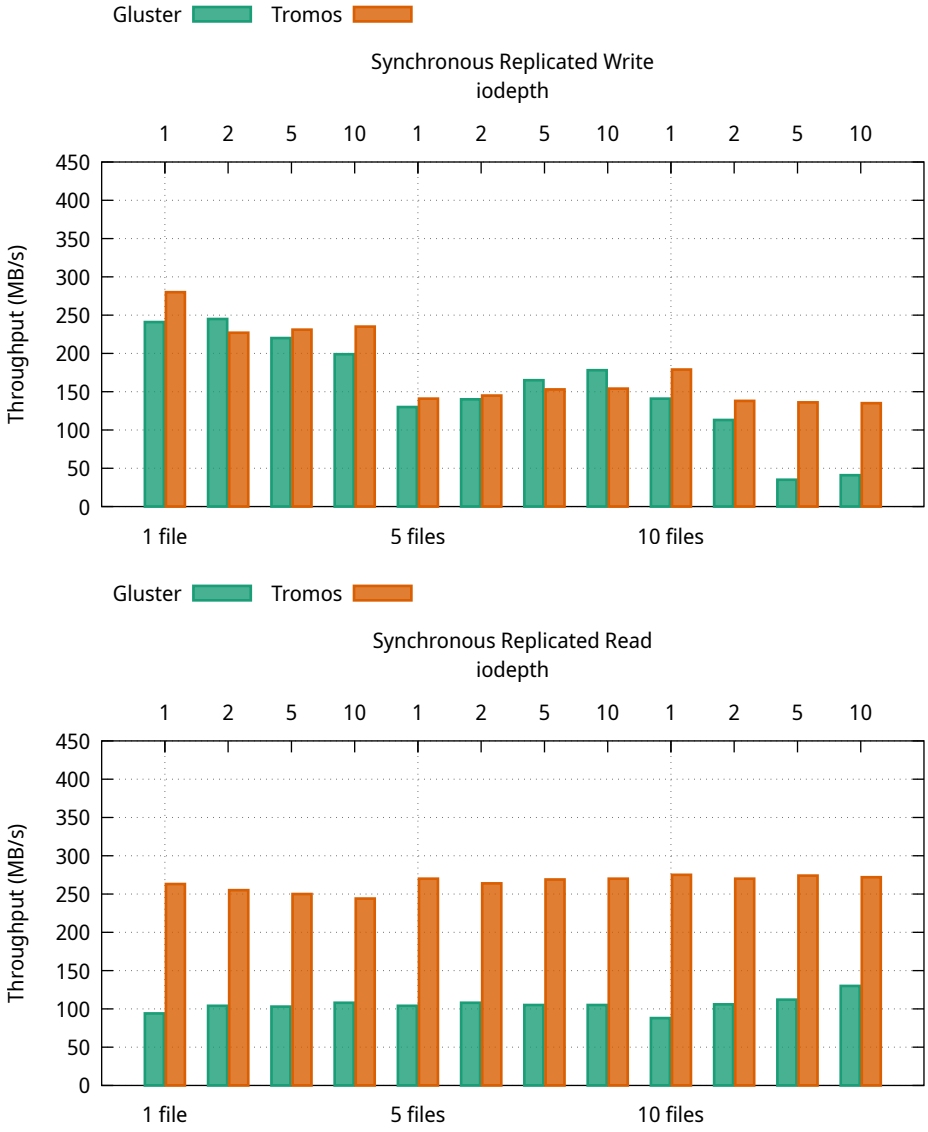

Figure 7.4 – Synchronous sequential I/O over replicated volume

7.2.4 Dispersed volume

In order to support stripping, all that it takes for *Tromos* is to apply the changes of Code 7.4 to the skeleton of Code 7.1. Figure 7.5 shows the measurements for reading and writing operations against a dispersed volume. The reason for the significantly low performance as compared to the previous volumes is that dispersed volumes are based on erasure-coding. Like any other redundancy coding, it is heavyweight, and therefore the I/O operations are CPU-bound.

```
1 ..............
2 ProcessingGrid:
3     "default":
4         plugin: "erasurecoding"
5         datablocks: 8
6         parityblocks: 3
7 ..............
```

Listing 7.4 – Integration of Erasure-coding processor, with 8 data blocks and 3 blocks for parity

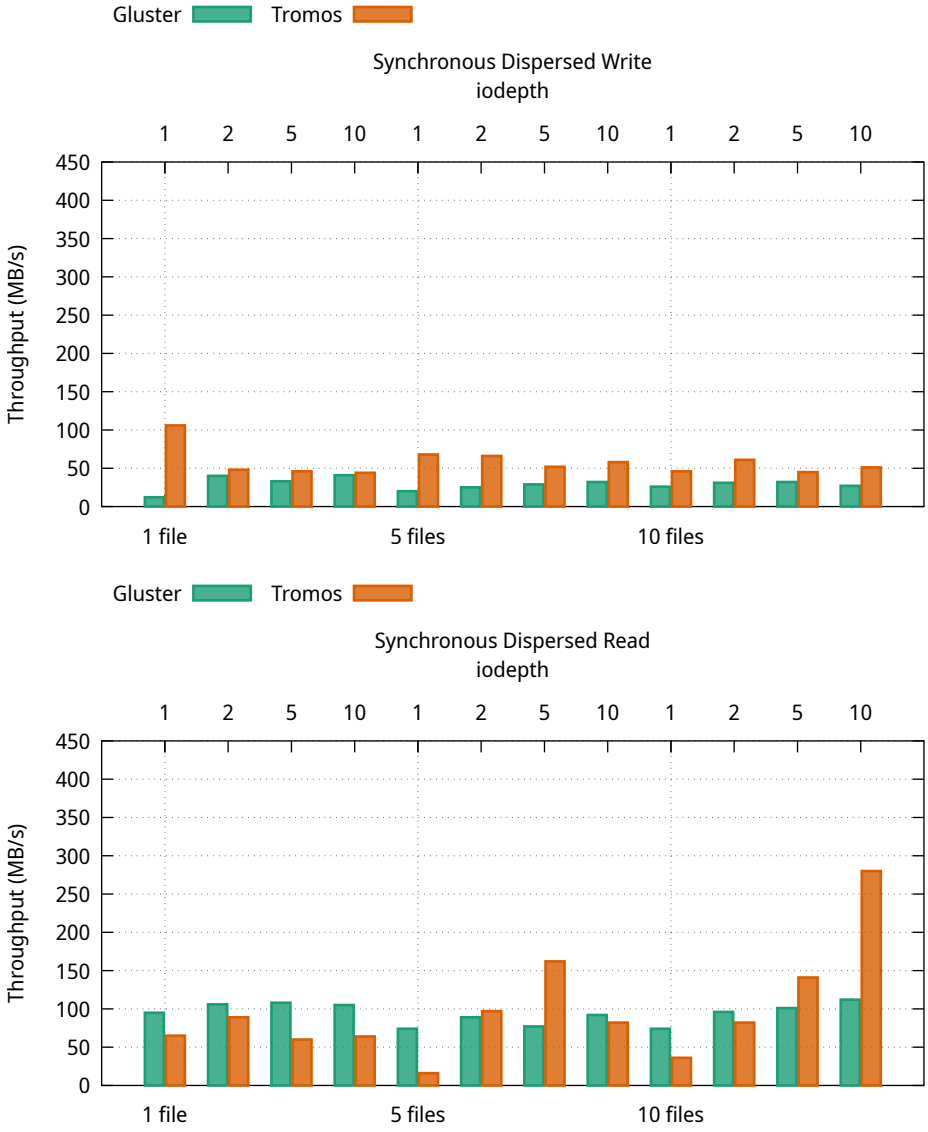

Figure 7.5 – Synchronous sequential I/O over dispersed volume

7.3 Asynchronous Workload

In this set of experiments, we evaluate the two systems against a write-many read-many access pattern. Such a pattern involves random and asynchronous I/O operations and mostly appears on data-management applications (e.g., databases). In the context of cloud computing, platforms that provide the backend for virtual machines and containers also exhibit the above pattern. In order to support this pattern, we had to apply certain modifications to *Tromos*. In this section, we present the encountered problems and our solutions.

To our surprise, when we tried the previous model against an asynchronous workload, we encountered the error of Figure 7.6. A more in-depth investigation showed that the error was perfectly reasonable. The *streaming* writer is append-only and therefore does not support random offsets. To counteract this limitation, we introduce another type of writer called *delta writer*. Compared to the *streaming* writer that handles all the application-requests as a single stream, the *delta* writer initiates a new stream of every incoming request. To integrate this new type of writer, we had to slightly modify the original skeleton with the changes of Code 7.5.

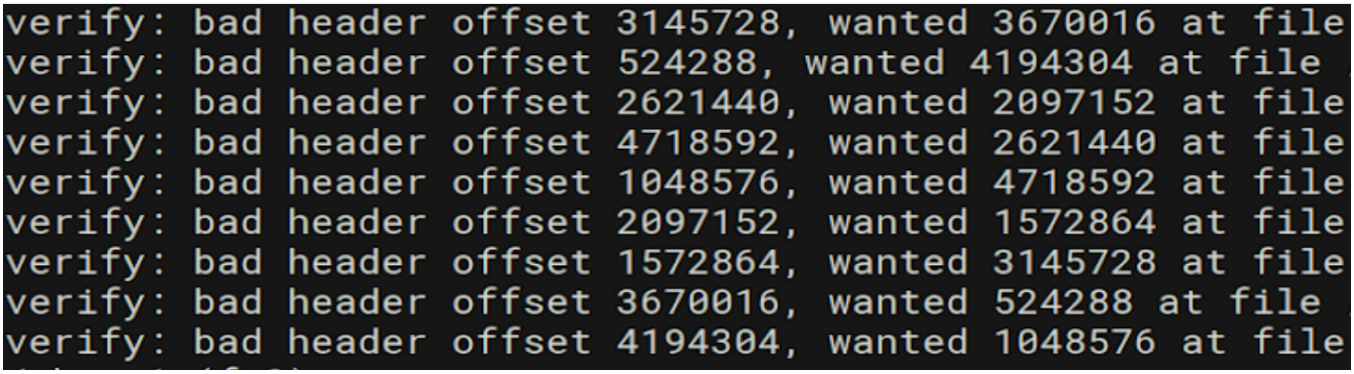

Figure 7.6 – Streaming writers are append-only: they do not support random I/O

```
1 Gateway:
2     plugin: fuse
3     writer: delta
4     mountpoint: /tmp/test
5 .........
```

Listing 7.5 – Integration of delta writer to support asynchronous and random I/O

Another strange behavior we encountered while trying to verify the written content was that every few iterations fio was returning an error about not finding the written file. A more in-depth investigation showed that on asynchronous setting the data verify operation was starting before the write operation was complete. As files had not been written at that time, verify was failing. To counteract, we employed a sequencer translator into the *Coordinator* to serialize the top-level requests. This translator exposes two parameters: blockw2r and blockw2w. The first blocks read operations while a write operation is running, and the second blocks a write operation while

another write operation is running. By selecting the proper combination, storage developers can adjust the desired consistency level. We integrated the translator into the storage line using the modifications of Code 7.6.

```
MessasingGrid:
    "H-range":
        Translator:
            "0":
                plugin: "sequencer"
                blockw2r: true
                blockw2w: false
        Connector: "boltdb"
            path: "/scratch/db/db0"
    "L-range":
        Translator:
            "0":
                plugin: "sequencer"
                blockw2r: true
                blockw2w: false
        Connector: "boltdb"
            path: "/scratch/db/db1"
Gateway:
    plugin: fuse
    writer: delta
    mountpoint: /tmp/test
.........
```

Listing 7.6 – Integration of Sequencing translator to Coordinators. blockw2r and blockw2w are arguments to control the desired consistency level

7.3.1 Access-pattern Visualization

Motivated by the consistency issued as described above, we also implement a translator for visualizing at real-time the incoming requests. Such translator makes it easier to reason about system's behavior. It also helps to eradicate hard-to-spot bugs related to concurrency and consistency issues. We integrated the translate into the storage line by applying the modifications of Code 7.7.

```
MessasingGrid:
    "H-range":
        Translator:
            "0":
                plugin: "sequencer"
                blockw2r: true
                blockw2w: false
            "1":
                plugin: "visualizer"
                path: "/tmp/tromos-h-visual"
        Connector: "boltdb"
            path: "/scratch/db/db0"
    "L-range":
        Translator:
            "0":
                plugin: "sequencer"
                blockw2r: true
                blockw2w: false
            "1":
                plugin: "visualizer"
                path: "/tmp/tromos-h-visual"
        Connector: "boltdb"
            path: "/scratch/db/db1"
Gateway:
    plugin: fuse
    writer: delta
    mountpoint: /tmp/test
.........
```

Listing 7.7 – Integration of Visualization translator to Coordinators

Figures 7.7 depicts the footprints of various workloads as seen by the *Coordinator*. On the top, from left to right, are the footprints of an append operation and a random write operations. On a first glance, the random write foot may seen peculiar since all writes begin and end at the same time. That is because the semantics of a *delta* writer as close-to-open. When a client opens a file handler is initiates a transaction with the *Coordinator*. When the client closes the file handler, it commits the transaction back to the *Coordinator*. In the meantime, the client can write directly to the *Devices* without the *Coordinator* participating in the operation. Thereby, the *Coordinator* is

only aware of the handler open and close times. A fine-grained solution would potentially involve timestamps on the write operations, but that would require a wall-clock between clients and *Coordinators*. Such a clock though would induce great synchronization overheads and therefore is omitted. The lower right plot shows the footprint of a file that is partially overwritten. The lower left plot shows the footprint of 4 repetitive experiments with 256 MB sequential transfers on 10 files.

Figure 7.8 is especially interesting, it depicts the footprint of random writes with synchronous data verification. Using the scheme of Figure 7.7, what happens is that when the visualizer receives a request is record the time and forwards it to the sequencer. When the sequencer replies, the translator records the return time. A line represents the start time and the end time of the request. If the sequencer blocks the request the end time from the start will significantly drift from each other. If it does not, the request will appear as a Dirac signal (almost vertical and instantaneous). The left graph represents the case of the visualizer located on top of the sequencer in the stack. In such case, the behavior aligns with the one explained above. When the visualizer is laying below the sequencer, it produces the right graph. Since there is no blocking layer between the visualizer and the connector, the visualizer senses the request handling as instantaneous. That is without compromising application correctness as sequencing indeed exists on the stack, but on a higher level than the visualizer and thereby it does not capture it. The above gives us a fundamental insight about Tromos usage; translator order does matter.

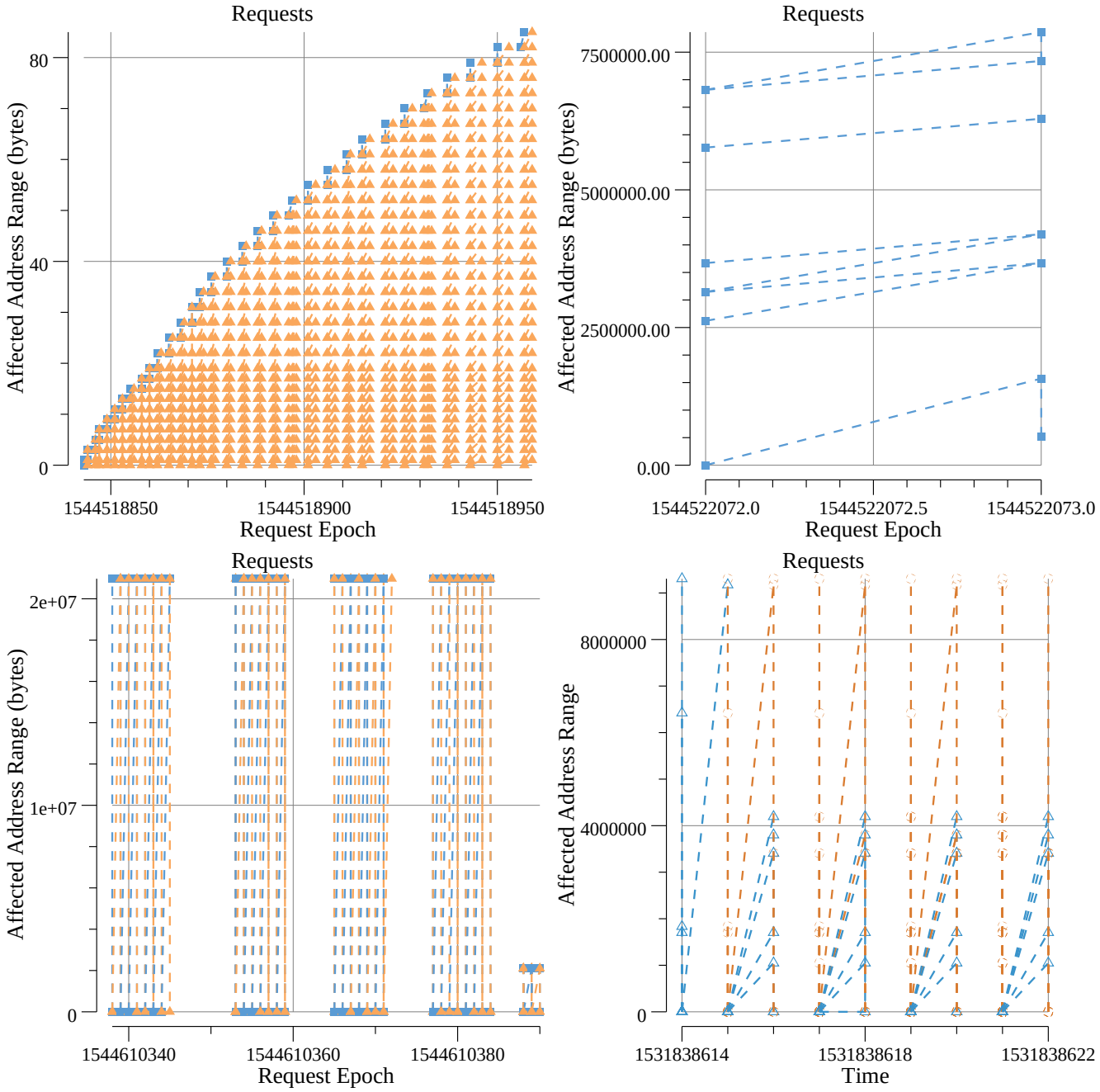

Figure 7.7 – Workload depictions. (top-left) append operations in a loop (top-right) write operations at random offsets (bottom-left) 4 repetitive experiments (bottom-left) 4 overwrites of the same file

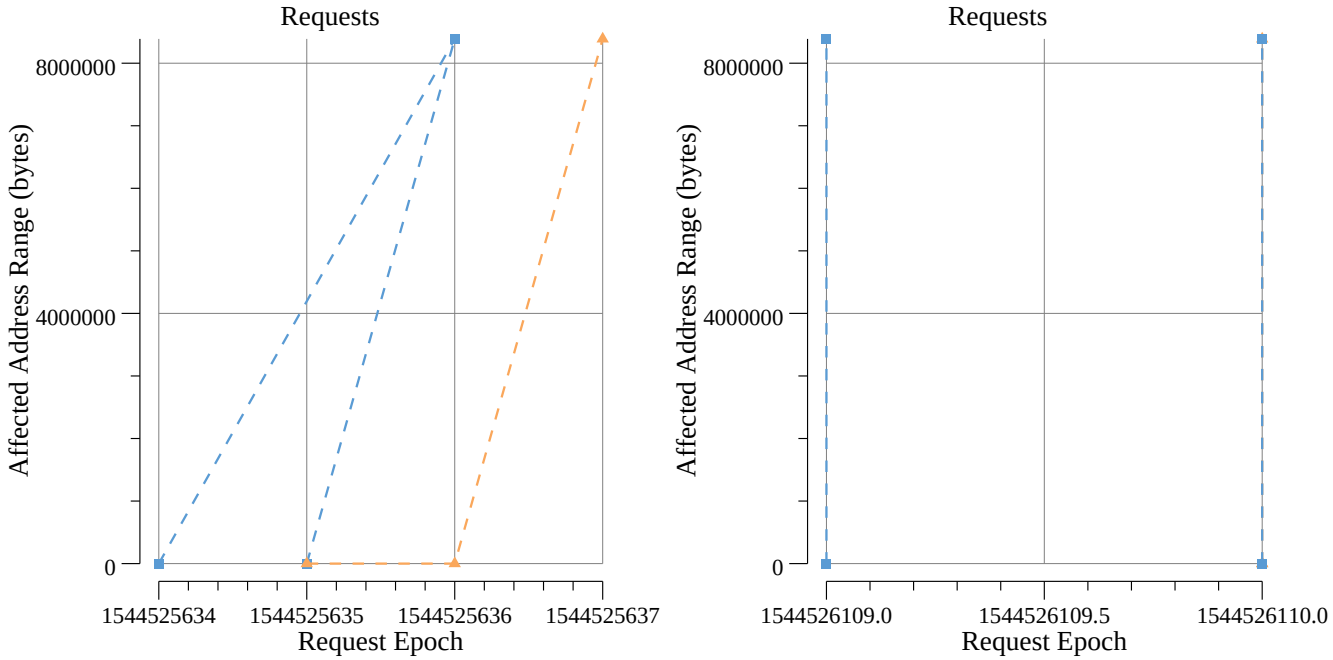

Figure 7.8 – Depiction of write-verify transaction. Left) The visualizer is located on top of the sequencer in the stack. Right) The visualizer is located below the sequencer in the stack

7.3.2 Burst Buffer

Counter-intuitively, the new writer was yielding significantly lower performance than the expected. Just a few MB/s. Further investigation revealed that the source of that erratic behavior was that fuse was breaking application requests into system requests. More specifically, fuse was invoking to *Tromos* as many requests as the number of blocks affected by the application-request. From the *Tromos* perspective, these system-requests are independent and it was handling them as such. Thereby, it was creating multiple small files on the backend, with size equal to the block size of fuse (Figure 7.9). Subsequently it was yielding significantly reduced performance and high IOPS on the backend, and low bandwidth utilization.

```
131072 /tmp/scratch/hdd3/root/88a3a25b.83
131072 /tmp/scratch/hdd3/root/88a3a25b.84
131072 /tmp/scratch/hdd3/root/88a3a25b.85
131072 /tmp/scratch/hdd3/root/88a3a25b.86
```

Figure 7.9 – Bottleneck in the Fuse gateway. The gateway breaks application requests to system requests equal to the fuse block size (128K). That triggers *Tromos* to create multiple small files on the backend

To counteract that deficit we implemented a burst-buffer translator for the devices. Its purpose is to aggregate small-sized requests into contiguous memory and collectively flush them when the channel closes (or when the memory buffer is full, whichever comes first). That allows to have only one physical file (binary large object) created on the backend, instead of multiple small-sized

physical files. In principle, the blob represents all the data written within an opened file handler, and write operations within that handlers are represented by logical files, at different offsets within the blob. Combined with *Coordinators* that include *Sequencer* translators, TromosFS exhibits behavior similar to a blob storage system with support for randomly seekable blobs with built-in transactions [76, 77, 109]. By applying the modifications presented in Code 7.8 we were able to run the asynchronous workload on Tromos.

```
1  ..............
2  DataGrid:
3      "n1d1":
4          Proxy:
5              plugin: "grpc"
6              uri: "10.200.0.1:7000"
7          Translators:
8              "0":
9                  plugin: "burstBuffer"
10                 blockSize: 10M
11         Connector:
12             plugin: "filesystem"
13             path: "/scratch/vol01"
14     ..............
```

Listing 7.8 – Integration of Burst buffer translator to Devices

7.3.3 Distributed Volume

Figure 7.10 shows the measurements for asynchronous read and write operations against a distributed volume. Notably, the write performance remains the same as with the synchronous workload. Compared to the *streaming* writer which enforces serialization of the application-requests, the *delta* writer handles every request independently and therefore allows for a higher degree of parallelization. Given that, we were expecting higher performance. However, performance improvement was not possible as requests were serialized on the burst buffer translator. A primary principle of Tromos is to make no assumptions as to the in-transit processing of data. The rest of the system should be oblivious to the I/O processor and the components it contains. As a result, there is no generic way to associate top-level input (as received by fuse gateway) to the output (as stored on the backend). Although it may be feasible for specific components (e.g., strip, mirror) or possible for others (e.g., block-based encryption), it cannot be generalized for all the possible components. Taking compression as reference, the relation between input to output depends on the data itself. Depending on the data entropy, the output may be more, less, or equal to the original input. It must be made clear that we do not refer to the data format, but the data size. Thereby, the only way to avoid overlaps and collision on the contiguous buffer of the burst buffer, the only way go is to impose serialization.

A second notable fact is that on read(), Gluster exhibits greater parallelism than *Tromos* as the

iopedth (or concurrently flying requests) increase. That, however, does not seem to be affected by the number of concurrently opened files. Investigation showed that the fuse implementation used for *Tromos* was serializing the operations occurred within an opened file handler. Different handlers could evolve separately.

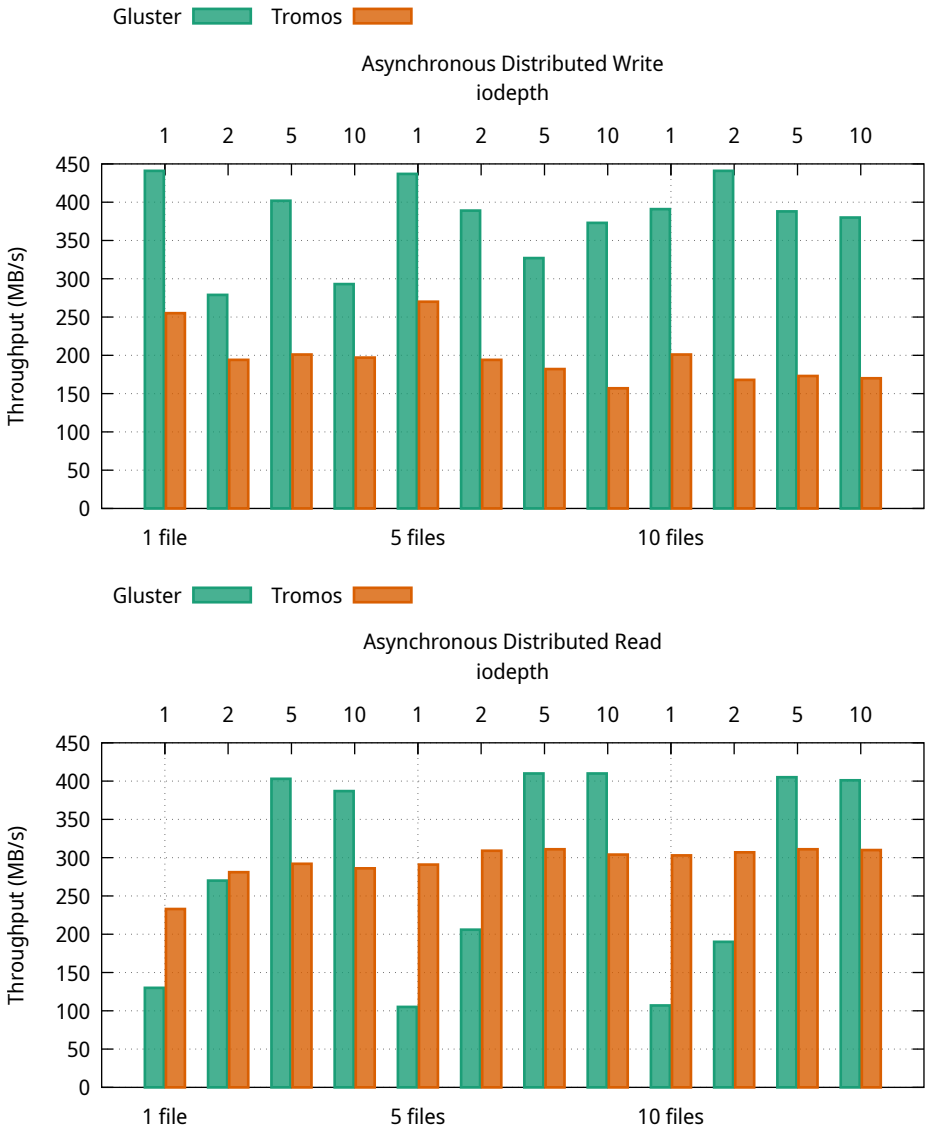

Figure 7.10 – Asynchronous random I/O over distributed volume

7.3.4 Stripped volule

Figure 7.10 shows the measurements for asynchronous read and write operations against a distributed volume. As one can see, Gluster failed to complete any operation, regardless of the number of concurrently opened files and the iodepth. Although we gave the potentials reasons of this erratic behavior on the previous section, we include the figure here for completeness.

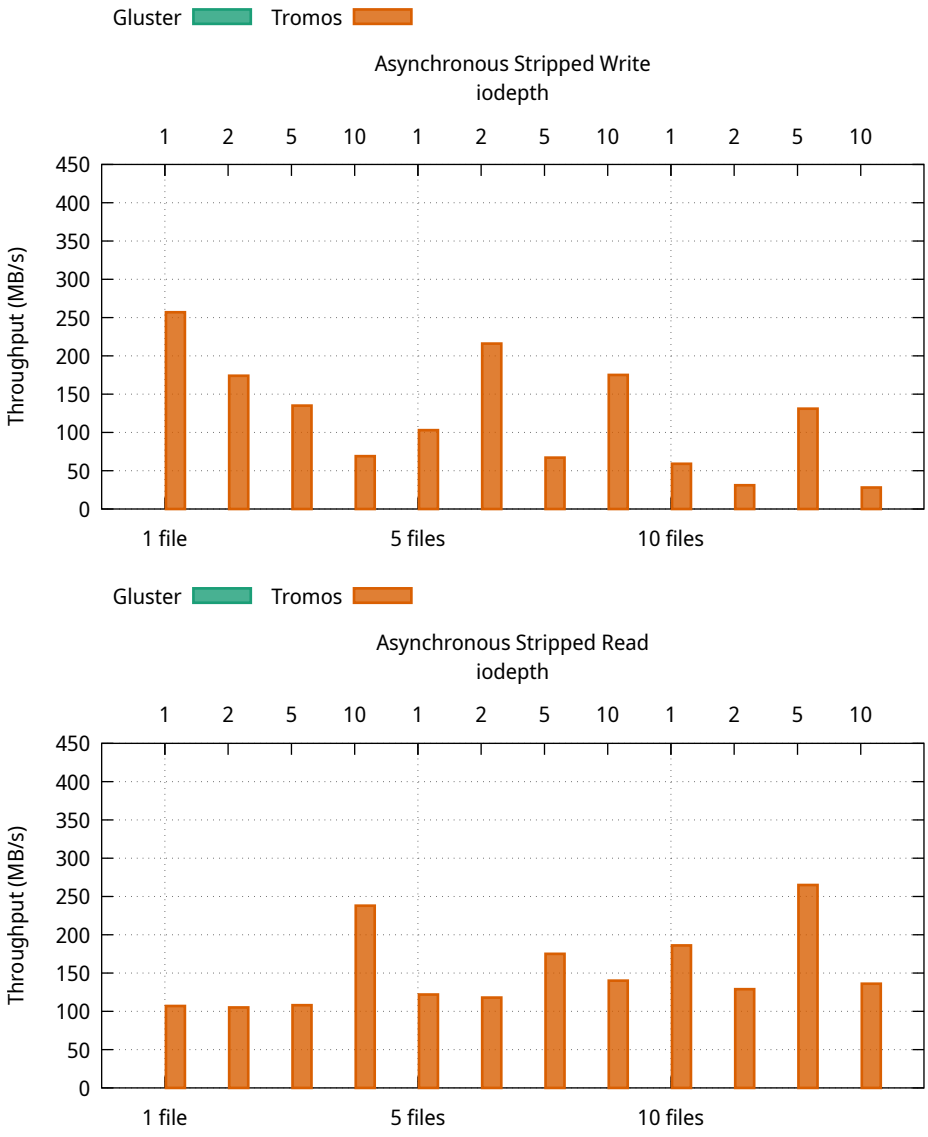

Figure 7.11 – Asynchronous random I/O over stripped volume

7.3.5 Replicated volume

Figure 7.10 shows the measurements for asynchronous read and write operations against a replicated volume. Combing the previous insights from the distributed volumes and synchronous stripped volume, we infer that the high degree of parallelism and the use of multiple devices can easily amortize the serialization overheads by the individual device.

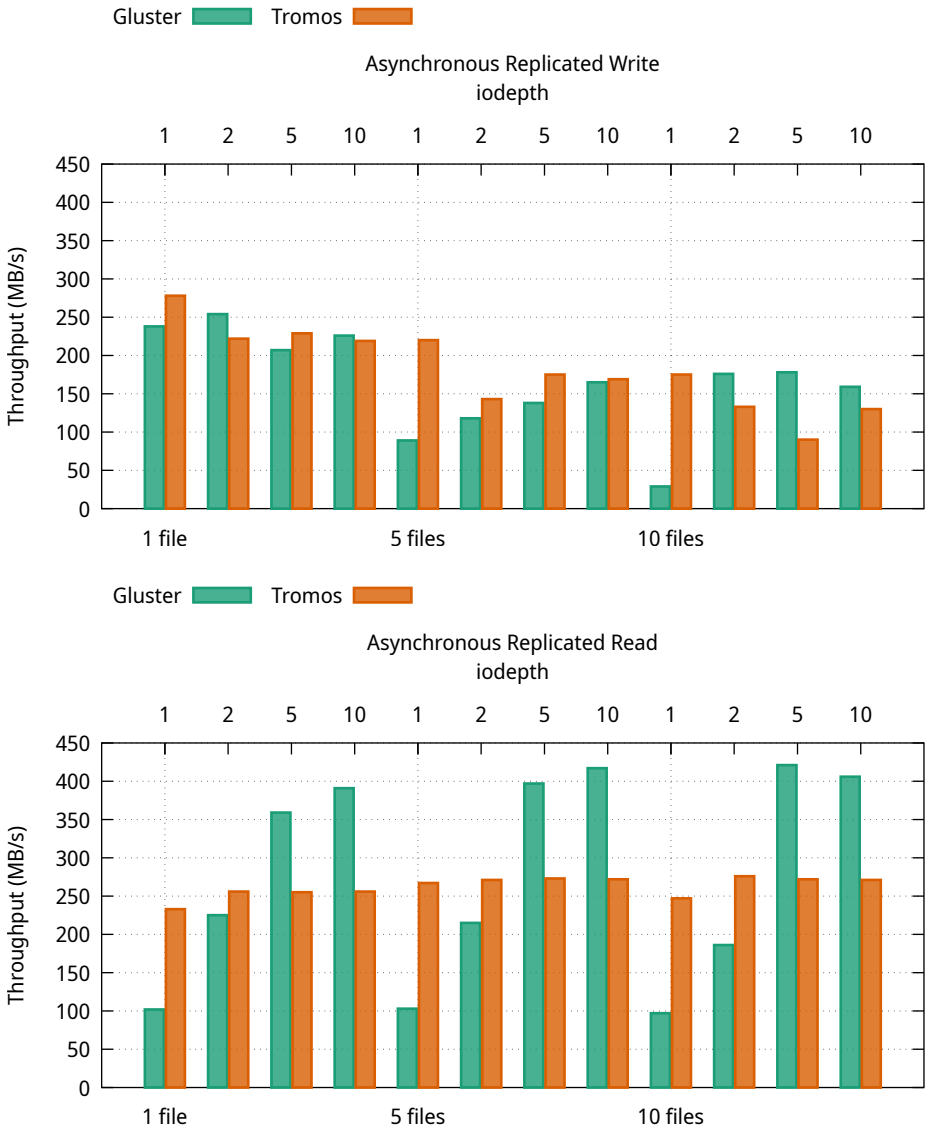

Figure 7.12 – Asynchronous random I/O over replicated volume

7.3.6 Dispersed volume

Dispersed I/O with regard to the number of concurrently opened files and iodepth for each file. Although CPU-bounded, dispersed volume seems to behave similarly to replicated volume. Especially the bottom-right figure shows that Gluster can scale better for a single process with multiple threads

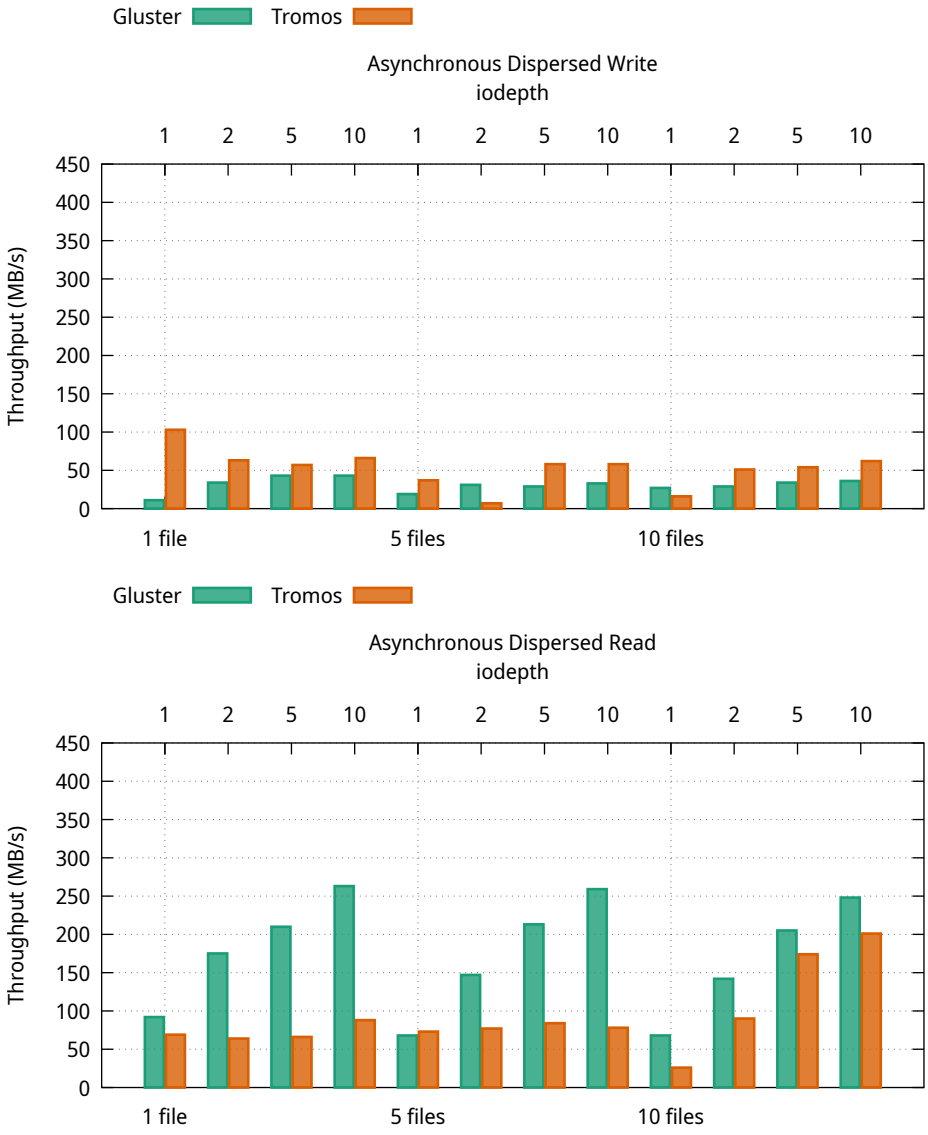

Figure 7.13 – Asynchronous random I/O over dispersed volume

7.4 Parallel Workload on scratchpad-storage

The second test of experiments is about scalability and parallel workloads. Although fio has helped to gain significant insights as to the *Tromos* behavior, its usage is limited to a single node. For this set of experiments, we used IOR (Interleaved or Random). It is a Parallel filesystem I/O

benchmark typically used to measure I/O performance on HPC environments [113].

Although the metric is still the throughput, as with fio experiments, the factors are different. Instead of using the number of files and iodepth, on this experiment we use workload class and parallel processes. In order to align our evaluation with typical workloads in an HPC environment, we take as reference the classes described in [64]. Nevertheless, since our evaluation testbed consists of fewer nodes than the ones described, we had to scale the workload to the size of our cluster, but still, retain the distinct characteristics of each class. Further, IOR will be run in a loop, doubling the number of processes per client node with every iteration from $SEQto- >$MAXPROCS. If SEQ=1 and MAXPROCS=8, then the iterations will be 1, 2, 4, 8 processes per node.

7.4.1 Scratchpad on computation nodes

A common issue of HPC application is that they produce multiple temporal files. Whether that be intermediate files or checkpoint files, they are transient data that will no longer be needed once the application has complete the computation. To avoid these temporary files “pollutting” and primary data-store of a data-center (e.g., lustre), this experiment is meant to illustrate the feasibility of joining the local storage of multiple computation-nodes into a distributed and customized scratchpad storage.

For this experiment, we used 5 hosts acting simultaneously as storage nodes and computational nodes. For every host, the MAXPROCS was set to 4 -equal to the number of CPU cores per machine-. Hence, in total, we evaluated parallel application spanning up to 20 parallel CPU cores. For sequential operation the results are shown in Figure 7.14 while for random operations the results are shown in Figure 7.15. Notably, Tromos can scale better than Gluster as the number of parallel processes increase. The reason is that it is adjusted to many of the unnecessary overheads imposed by a strongly POSIX-compliant filesystem like Gluster. However, Tromos significantly lacks behind Gluster on read. The difference is so great that could not be attributed to anything else rather than local caching. That was reasonable since computational nodes were also the storage nodes, and therefore is to highly like for data to be local on some nodes. In order to verify this theory we conducted the next set of experiments.

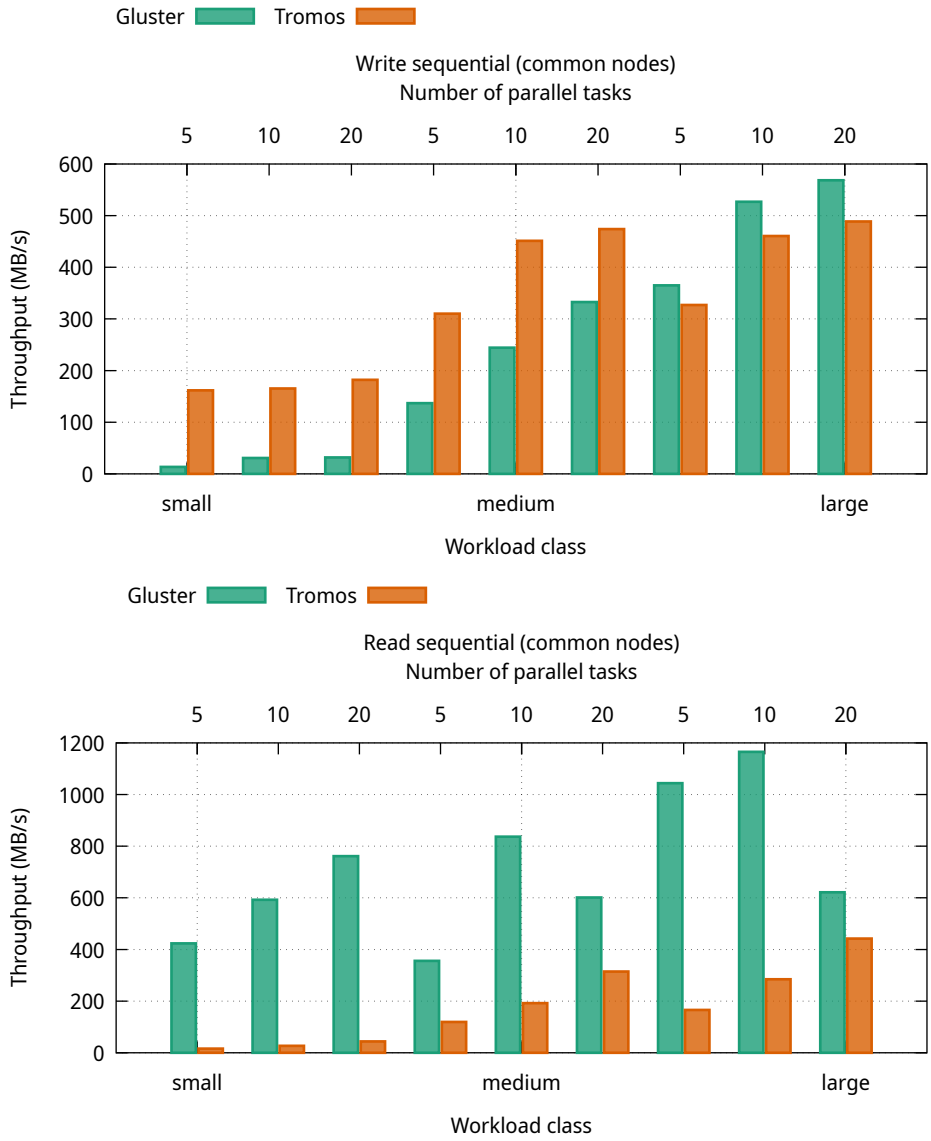

Figure 7.14 – Parallel sequential workload, with the storage nodes being the same than the computation nodes

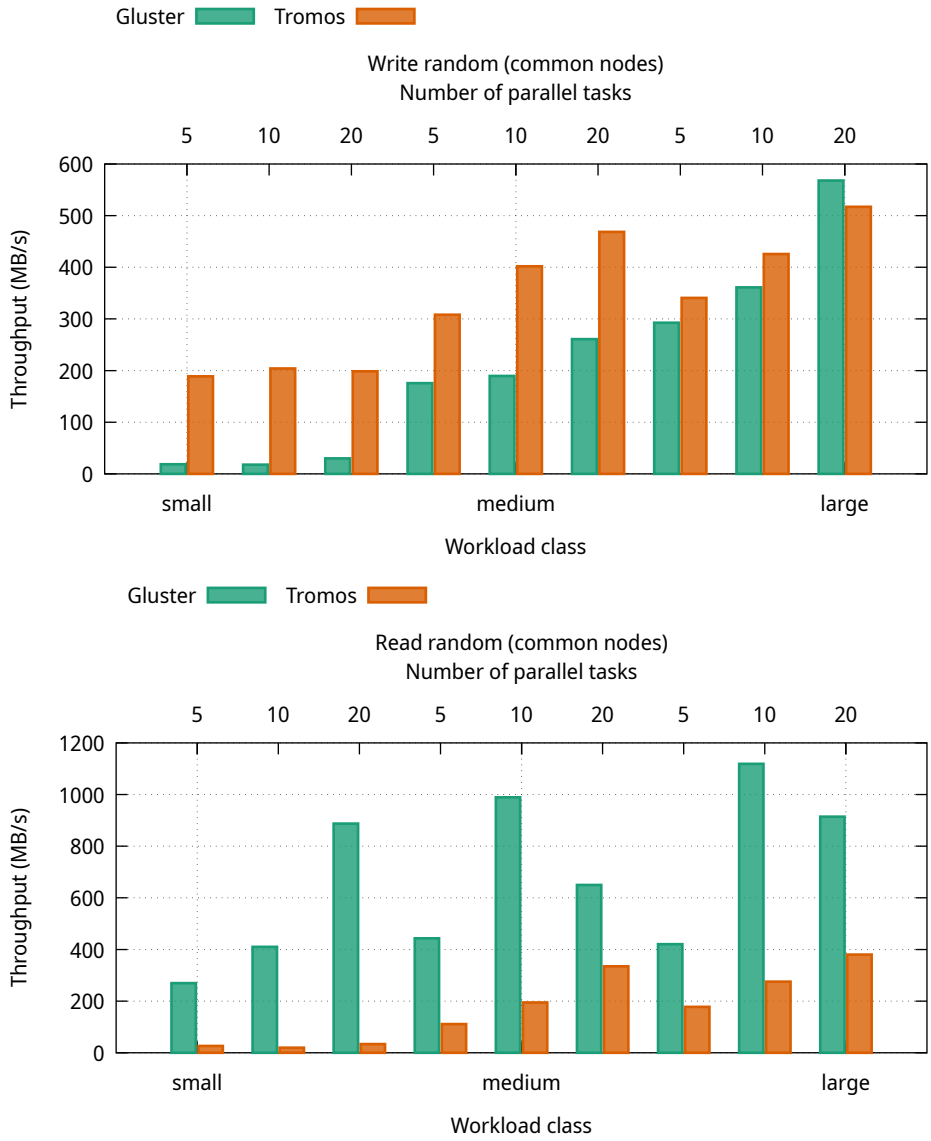

Figure 7.15 – Parallel random workload, with the storage nodes being the same than the computation nodes

7.4.2 Scratchpad on storage nodes

This experiment is to its greater extend the same as before, with the only difference that we separate the computation nodes from the nodes storage. The reason is to make the comparison between the two systems more objective. The results are shown in Figure 7.16 for sequential workload and 7.17 for random workload. Interestingly, Tromos performance remained stable while Gluster dropped by half. Investigation showed that the reason was the way Gluster and Tromos handles writing on local resources. When the destination is a locally attached disk, Gluster performs the write directly to that disk using the system calls. *Tromos* lack this optimization and all the operations go through the RPC mechanism. In turn, that means that all the operation must go through the network stack and sustain any imposed overhead. It is in our future plans to include a similar mechanism to differentiate the way *Tromos* access local *Devices* from remote *Devices*.

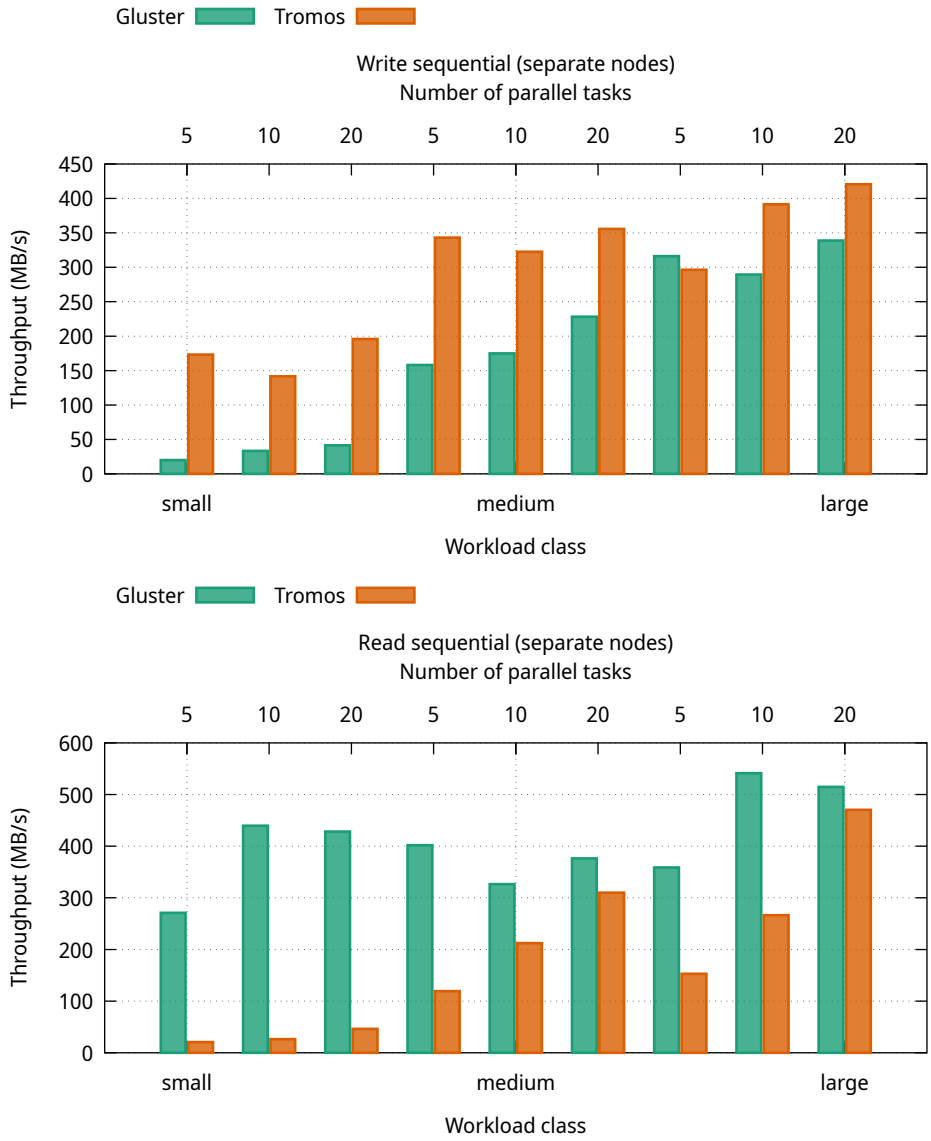

Figure 7.16 – Parallel sequential workload, with the storage nodes being different than the computation nodes

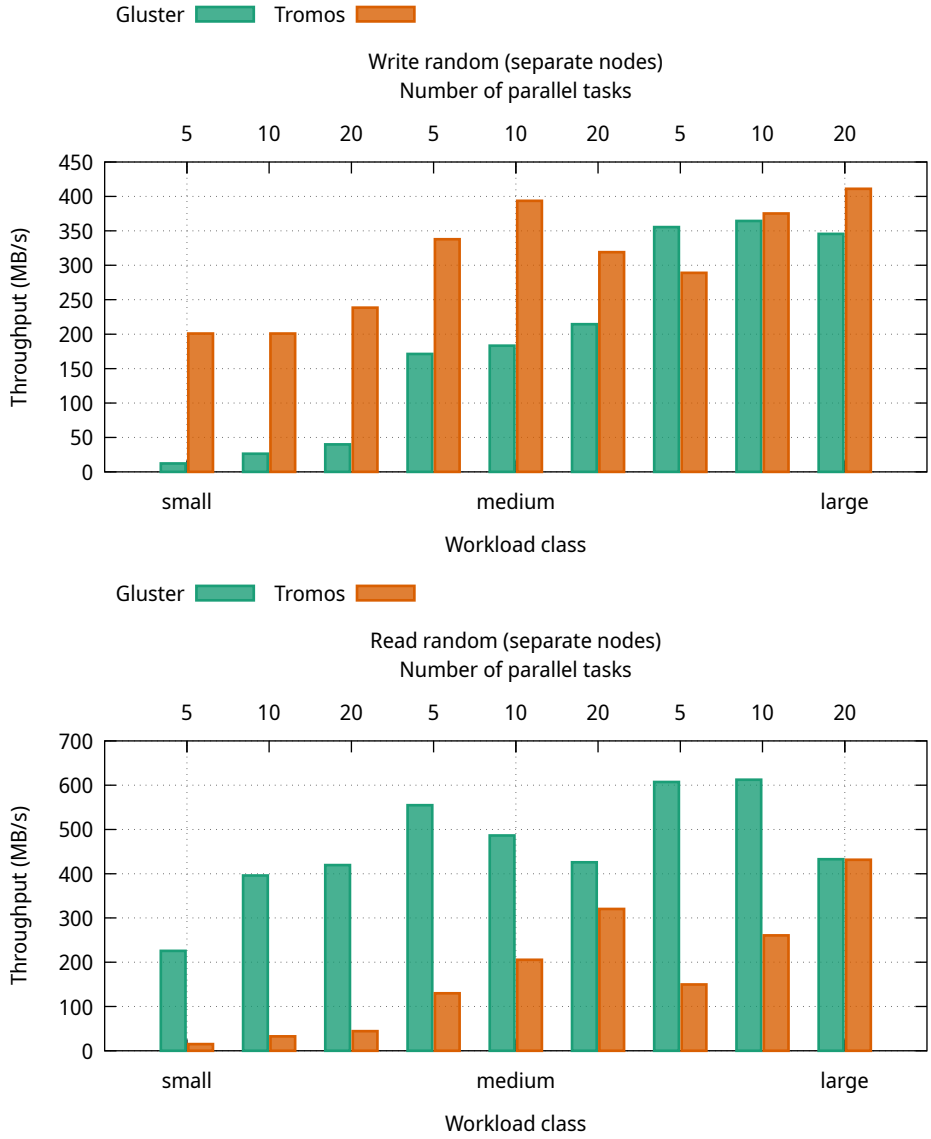

Figure 7.17 – Parallel random workload, with the storage nodes being different than the computation nodes

Streamer

The difference between this setting and the previous is the use of streaming writer instead of delta writer. As Figure 7.18 shows, streamer can yield significantly better results by trading throughput for random-access. That, however, does not necessarily pose a strict limitation since most of the HPC workloads tend to be small and sequential [52].

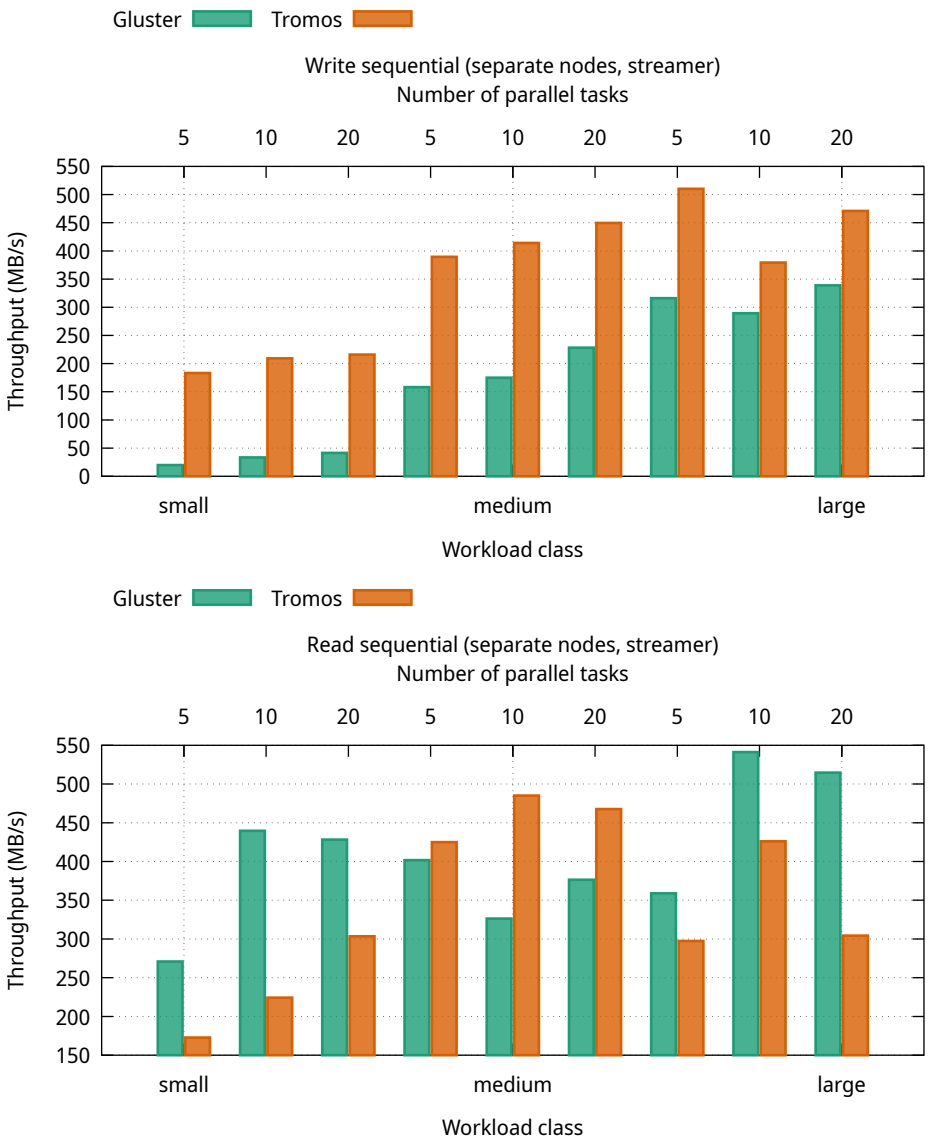

Figure 7.18 – Usage of streaming instead of delta writer for parallel sequential workload. Separated computational and storage nodes.

Chapter 8

Conclusion

In this dissertation, we introduced “application-tailored” storage systems as the next logical in the storage evolution. Conventional “general-purpose” storage systems provide applications with capacity, performance, and data protection, but cannot provide data-management specific to the individual application. Oppositely, “domain-specific” systems can provide applications with domain-specific data-management, but cannot adjust to dynamically changing applications without the direct intervention of the administrator. “Programmable storage” are extensions to general-purpose systems that allow applications to communicate at runtime their requirements and I/O hints, without any human intervention. They cannot handle, however, applications whose requirements go beyond the scope of a single data-store. Our proposal, “application-tailored” storage, is a data-management middleware that is a peer with the application (intimate knowledge), and leverage arbitrary number of third-party data-stores to persistently store the data. The middleware separates the application logic from the I/O logic, hence separating changes made to application codes by science users from changes made to I/O actions by developers or administrators. Such design helps to defer I/O decisions until the deployment phase, which is the key for portability; storage federation over non-collaborative storage vendors; second-order scaling independently to the data-stores; and policy-based data distribution with criteria such as resiliency, performance, storage efficiency, security, and cost. That, however, requires the developers to design and implement a new middleware, customized to the specific application.

Building that middleware is non-trivial, requires a lot of effort and programming skills, and is error-prone. To this end, we proposed *Tromos*; a framework for assembling “lego” components into customized storage systems, and tooling to build, test and deploy artifacts for these assemblies. *Tromos* provides a repository of pluggable components for developers to define policies regarding storage aspects (e.g., atomic transactions, consistency level, resource abstraction, resource management) and data management (e.g., in-transit processing, placement). A provided declarative language allows the developers to model their target storage environment as compositions of those components, without asking for any systems programming. The framework fabricates, deploy, and handle that environment, transparently to the application. To make

an analogy, if the middleware is the storage-equivalent of a container, *Tromos* is the storage-equivalent of *Docker*. Architecturally, *Tromos* lays on four main pillars i) declarative languages and pluggable components for composing service drivers ii) services for abstracting the external data-stores, the data transformation and distribution, and the consistency and the metadata handling iii) client-side middleware for building virtual storage infrastructures using the primitives of the services iv) client-side libraries and gateway for abstracting the low-level API into an API friendlier to developers.

As a proof-of-concept, we used our framework to fabricated various data-management middleware that we compared against Gluster; a “general purpose” distributed filesystem owned by Redhat.We experimented with a broad set of various application workloads (synchronous, asynchronous, parallel, sequential, random) and backend functionality (distribution, replication, stripping, erasure-coding). These experiments gave us insights about the capabilities and limitations of *Tromos*, but most importantly, they proved that *Tromos* makes possible to implement at least the same functionality as to an existing system, without systems programming. They also proved that application that runs on top customized middleware, with carefully included components, can experience greater performance than running on top of general-purpose storage systems. Hence, although *Tromos* is a young project, it could supersede Gluster in performance by removing any unnecessary functionalities from the storage-line. In general, the principle behind “application-tailored storage” is: if specific functionality is needed, add it as pluggable policy. If it is not needed, omit it to avoid the overhead.

8.1 Future Direction

Perhaps the most intriguing usage of *Tromos* is as a toolkit intended for engineers, researchers and enthusiasts looking to modify, hack, fix, experiment, invent and build distributed storage systems. Not for those seeking for a commercially supported “black-box” system, but for those who want to work and learn with open source code. In other words, to be an in-vitro emulator for research in distributed systems. With *Tromos*, researchers will be able to easily experiment with new algorithms on a running system without the burdens and waste of timing of browsing into the source to find where to place to their modifications. Adding a new placement should be as easy as building a plugin. Such an approach would further advocate more objective comparison (plugins minimize side effects) and would promote system reproducibility and behavior reasoning. Although similar tools such as Docker and Terraform pre-exist, they are focused on the application-level and the infrastructure-level, respectively. To the best of the author knowledge, *Tromos* is the first tool that simplifies the development of complex storage environments into an intuitive language.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf. Extending i/o through high performance data services. In *Cluster Computing*, Austin, TX, September 2007. IEEE International.

[3] H. Abbasi, J. Lofstead, F. Z. F. Zheng, K. Schwan, M. Wolf, and S. Klasky. Extending I/O through high performance data services. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[4] Amazon. Aws storage services overview, 2016.

[5] Amazon. Amazon s3, 2018.

[6] A. F. Anta, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing distributed ledger objects. *SIGACT News*, 49(2):58–76, June 2018.

[7] Apache. Apache HBase – Apache HBase™ Home, 2015.

[8] Apache. Apache bookkeeper, 2018.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[10] axboe. Flexible i/o tester, 2018.

[11] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[12] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, Dec. 2013.

[13] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[14] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, K. Antypas, and Prabhat. Accelerating Science with the NERSC Burst Buffer Early User Program. *Proceedings of the 2016 Cray User Group*, pages 1–15, 2016.

[15] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, Nov. 1987.

[16] E. Blossom. Gnu radio: Tools for exploring the radio frequency spectrum. *Linux J.*, 2004(122):4–, June 2004.

[17] BoltDB. An embedded key/value database for go., 2016.

[18] M. Borland. User's guide for sdds toolkit, 2017.

[19] P. J. Braam. The Lustre Storage Architecture. *Change*, 23:1–23, 2004.

[20] J.-C. Brendel. The benefit of hybrid drives, 2014.

[21] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.

[22] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):1–42, 2009.

[23] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[24] Canonical. What are the different types of storage: block, object and file?, 2015.

[25] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[26] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[27] J. Daemen and V. Rijmen. Aes proposal: Rijndael, 1999.

[28] S. G. Dean, Jeffrey; Ghemawat. Map-Reduce, 2004.

[29] C. Doxsey. *Introducing Go. Build Reliable, Scalable Programs*. O'Reilly Media, 1 edition, 1 2016.

[30] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[31] G. Eisenhauer. The evpath library, 2015.

[32] Facebook. Smaller and faster data compression with zstandard, 2017.

[33] I. Fette and A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[34] A. Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.

[35] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[36] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[37] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[38] G. Goodson. NFSv4 pNFS Extensions. Internet-Draft draft-ietf-nfsv4-pnfs-00, Internet Engineering Task Force, Oct. 2005. Work in Progress.

[39] Google. a fast and lightweight key/value database library by google., 2009.

[40] Google. Flatbuffers: a memory efficient serialization library, 2014.

[41] Google. grpc remote procedure call, 2016.

[42] Google. Limits and quotas, 2017.

[43] Google. Tensorboard: Visualizing learning, 2018.

[44] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, W. Oppermann, and P. Michiardi. Iostack: Software-defined object storage. *IEEE Internet Computing*, 20(3):10–18, May 2016.

[45] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA, 2016. USENIX Association.

[46] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[47] HashiCorp. Consul, 2016.

[48] HashiCorp. Write, plan, and create infrastructure as code, 2018.

[49] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005.

[50] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *In Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, 1997.

[51] D. Hildebrand and A. Nisar. pNFS, POSIX, and MPI-IO: a tale of three semantics. *of the 4th Annual Workshop on*, pages 32–36, 2009.

[52] D. Hildebrand, L. Ward, and P. Honeyman. Large files, small writes, and pnfs. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 116–124, New York, NY, USA, 2006. ACM.

[53] hprose. High performance remote object service engine, 2017.

[54] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[55] F. Isaila, J. Carretero, and R. Ross. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, pages 346–355, 2016.

[56] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe. Making the case for reforming the i/o software stack of extreme-scale systems. *Advances in Engineering Software*, 111:26 – 31, 2017. Advances in High Performance Computing: on the path to Exascale software.

[57] M. R. Jain. Introducing badger: A fast key-value store written purely in go, 2017.

[58] A. jclouds. The java multi-cloud toolkit, 2017.

[59] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1561–1570. IEEE, 2017.

[60] juju. Efficient token-bucket-based rate limiter package, 2015.

[61] E. Kakoulli and H. Herodotou. Octopusfs: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 65–78, New York, NY, USA, 2017. ACM.

[62] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. 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, pages 654–663, New York, NY, USA, 1997. ACM.

[63] G. Koloventzos, R. Nou, A. Miranda, and T. Cortes. Hetfs: A heterogeneous file system for everyone. In *ISC Workshops*, 2017.

[64] L. L. N. Laboratory. Ior: I/o performance benchmark, 2018.

[65] Libcloud. Python library which hides differences between different cloud provider api, 2017.

[66] LinuxFoundation. Open software defined storage, 2016.

[67] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, 1988.

[68] G. Lockwood. What's so bad about posix i/o?, 2017.

[69] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton. Daos and friends: A proposal for an exascale storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 50:1–50:12, Piscataway, NJ, USA, 2016. IEEE Press.

[70] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[71] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud.

[72] M. Lowicki. Interfaces in go, 2017.

[73] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):5:1–5:28, Mar. 2017.

[74] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*, pages 773–785, 2017.

[75] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, 2014. USENIX Association.

[76] P. Matri, Y. Alforov, A. Brandon, M. Kuhn, P. Carns, and T. Ludwig. Could blobs fuel storage-based convergence between hpc and big data? In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 81–86, Sept 2017.

[77] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. S. Pérez. Tyr: Blob Storage Meets Built-In Transactions. In *IEEE ACM SC16 - The International Conference for High Performance Computing, Networking, Storage and Analysis 2016*, Salt Lake City, United States, Nov. 2016.

[78] Maxta. Maximize the promise of hyper-convergence, 2016.

[79] S. McKee, E. Kissel, B. Meekhof, M. Swany, C. Miller, and M. Gregorowicz. Osiris: a distributed ceph deployment using software defined networking for multi-institutional research. *Journal of Physics: Conference Series*, 898(6):062045, 2017.

[80] G. Memik, M. T. Kandemir, W.-K. Liao, and A. Choudhary. Multicollective i/o: A technique for exploiting inter-file access patterns. *Trans. Storage*, 2(3):349–369, Aug. 2006.

[81] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[82] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos. Knowledge-defined networking. *SIGCOMM Comput. Commun. Rev.*, 47(3):2–10, Sept. 2017.

[83] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[84] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[85] P. Mishra and A. Somani. Host managed contention avoidance storage solutions for big data. *Journal of Big Data*, 4:18, 12 2017.

[86] J. Molina. Apache HIVE. *Hadoop Magazine*, 1(1):8–16, 2014.

[87] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[88] J. P. Morrison. Data stream linkage mechanism. *IBM Systems Journal*, 17(4):383–408, 1978.

[89] Mulesoft. Multi resource transaction, 2016.

[90] F. Nikolaidis, N. Kossifidis, T. Leibovici, and S. Zertal. Towards a transparent i/o solution. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1221–1228, May 2018.

[91] NoFlow. Noflo is a javascript implementation of flow-based programming (fbp), 2017.

[92] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight i/o for scientific applications. In *2006 IEEE International Conference on Cluster Computing*, pages 1–11, Sept 2006.

[93] Oracle. 128-bit storage: are you high?, 2004.

[94] OrangeFS. The orangefs project, 2017.

[95] D. Petcu, C. Craciun, and M. Rak. Towards a cross platform cloud api - components for cloud federation., 01 2011.

[96] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, 2014. USENIX Association.

[97] A. Rajasekar, R. Moore, C.-y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, and B. Zhu. *iRODS Primer: Integrated Rule-Oriented Data System*. Morgan and Claypool Publishers, 2010.

[98] I. Red Hat. Easy to use local storage management for linux, 2018.

[99] I. Red Hat. A management tool for virtual and private cloud infrastructure, 2018.

[100] RedHat. Glusterfs documentation, 2016.

[101] Redhat. The device mapper, 2018.

[102] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[103] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press.

[104] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[105] T. M. Ruwart. Osd: A tutorial on object storage devices. In *19th IEEE Symposium on Mass Storage Systems and Technologies*, 2002.

[106] S. Sahu, P. Nain, C. Diot, V. Firoiu, and D. Towsley. On achievable service differentiation with token bucket marking for tcp. *SIGMETRICS Perform. Eval. Rev.*, 28(1):23–33, June 2000.

[107] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases*, 3:325–360, 10 1995.

[108] Seagate. The seagate kinetic open storage vision, 2015.

[109] R. Sears, C. V. Ingen, and J. Gray. To BLOB or not to BLOB: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168*, pages 1–11, 2007.

[110] M. A. Sevilla, I. Jimenez, N. Watkins, J. LeFevre, P. Alvaro, S. Finkelstein, P. Donnelly, and C. Maltzahn. Cudele: An api and framework for programmable consistency and durability in a global namespace. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, volume 00, pages 960–969, May 2018.

[111] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. Lefevre, and C. Maltzahn. Malacology: A Programmable Storage System. In *EuroSys '17*, pages 978–1, 2017.

[112] N. Shalom. Aria and the 'least common denominator' problem of cloud portability, 2016.

[113] H. Shan and J. Shalf. Using ior to analyze the i/o performance for hpc platforms. In *In: Cray User Group Conference (CUG'07*, 2007.

[114] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. Padua. Performance portability with the chapel language. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 582–594, Washington, DC, USA, 2012. IEEE Computer Society.

[115] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 33–, New York, NY, USA, 2003. ACM.

[116] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sroute: Treating the storage stack like a network. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 197–212, Santa Clara, CA, 2016. USENIX Association.

[117] Stow. Cloud storage abstraction package for go, 2017.

[118] M. Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 1st edition, 2015.

[119] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Sept 1994.

[120] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, 2002.

[121] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[122] L. Tianhua, Z. Hongfeng, C. Guiran, and Z. Chuansheng. The design and implementation of zero-copy for linux. In *2008 Eighth International Conference on Intelligent Systems Design and Applications*, volume 1, pages 121–126, Nov 2008.

[123] N. H. Tolia, M. A. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *USENIX Annual Technical Conference, General Track*, 2003.

[124] Trustmaster. Goflow - dataflow and flow-based programming library for go. https://github.com/trustmaster/goflow, 2013.

[125] K. Ueda, J. Nomura, and M. Christie. Request-based Device-mapper multipath and Dynamic load balancing. *Linux Symposium*, pages 235–243, 2007.

[126] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[127] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 16:1–16:14, New York, NY, USA, 2014. ACM.

[128] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, UK, 2002. Springer-Verlag.

[129] O. Weidner, M. Atkinson, A. Barker, and R. Filgueira Vicente. Rethinking high performance computing platforms: Challenges, opportunities and recommendations. In *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, DIDC '16, pages 19–26, New York, NY, USA, 2016. ACM.

[130] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[131] Z. Wilcox-O'Hearn and B. Warner. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 21–26, New York, NY, USA, 2008. ACM.

[132] Workiva. go-datastructures, 2016.

[133] Z. H. Wu. A log-structured file system based on leveldb. In *Advanced Manufacturing and Information Engineering, Intelligent Instrumentation and Industry Development*, volume 602 of *Applied Mechanics and Materials*, pages 3481–3484. Trans Tech Publications, 10 2014.

[134] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[135] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, 1993.

[136] Q. Zheng, K. Ren, and G. Gibson. Batchfs: Scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, pages 1–6, Piscataway, NJ, USA, 2014. IEEE Press.

[137] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider. Deltafs: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, pages 1–6, New York, NY, USA, 2015. ACM.