Infusing Pub-Sub Storage with Transactions

Liana V. Rodriguez[†] John Bent[‡] Tim Shaffer[‡] Raju Rangaswami[†]
[†] *Florida International University* [‡] *Seagate Technology*

ABSTRACT

The need to support new features in existing storage systems is an ongoing concern for storage developers. So is the desire to develop next generation storage systems that can adopt newly developed feature improvements with relative ease. Extending storage systems is challenging because of the inherent complexity of their codebases and the need to ensure that the storage state does not become corrupt or inconsistent when enabling new features. In this work, we examine a new storage architecture, FDMI, that uses the well-established publish-subscribe model for extending the feature set of a host storage system using *plugins*. A central mechanism in FDMI is *transactional coupling*. With transactional coupling, the subscribed plugin can either create new transactions that execute asynchronously following the successful completion of the precipitating event or can participate in the pending transaction and control whether the precipitating event itself will or will not be committed. We further create a classification of transactional mechanisms as well as possible desired plugin functionality and explore the matrix of these two classifications to create a new model for faster, safer distributed storage development.

ACM Reference Format:
Liana V. Rodriguez[†] John Bent[‡] Tim Shaffer[‡] Raju Rangaswami[†], [†] *Florida International University* [‡] *Seagate Technology* . 2022. Infusing Pub-Sub Storage with Transactions. In *Proceedings of 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.475/1234.5678

1 INTRODUCTION

Storage systems are at the intersection of multiple technology trends. Their designs are expected to evolve to address the massive growth in data volume from emerging applications and to adopt new advances from research and practice, while also adapting to emerging storage device technology. The large number of storage system improvements over the years reflects this trend [3, 8, 16, 18, 21, 22, 34, 44, 46, 49].

Architecture	Storage access	Transactional guarantees	No I/O Amp	Indep. scale	Ease dev.
Integrated	✓	✓	✓	✗	✗✗✗
Interposed I/O	✓	✗	✓	✓*	✗
Conv. Pub-Sub	✗	✗	✗	✓	✓
FDMI	✓	✓	✗	✓	✓

Table 1: Properties of extensible storage system architectures. *The * indicates that some systems (e.g., ABLE [17]) are exceptions, while "✗✗✗" indicates significantly higher level of said property than a single "✗".*

Implementing new features today requires a tight coupling with the storage system itself, requiring an in-depth understanding of its codebase, matching its performance properties, as well as ensuring that state changes do not introduce corruption or inconsistency. The conventional approach to introducing a new feature is *integrated* development which requires developers with a deep understanding of the existing storage system codebase to natively integrate the new feature. The drawbacks of this approach are the significant development time, effort, required expertise, and an inability to scale the resources used for the new feature independently. Recognizing these limitations, researchers have created extensible harnessing for storage systems that allows new features to be *interposed* within the storage stack without modifying the existing storage system [1, 4, 15, 17, 38, 50]. The interposed *feature layer*, however, is in the storage I/O path handling every client I/O request, and therefore must meet stringent performance requirements. This in itself requires a careful design to account for the interactions between the feature layer and the storage system, increasing development complexity. Furthermore, independent scaling is only possible if the I/O traffic to the interposed feature layer gets redirected to independent hardware resources which further compromises the performance of storage client operations.

An alternative approach is a *loosely coupled* development and deployment model wherein the storage feature is added outside of the primary I/O path using a narrow API exposed by the storage system. Our proposal, FDMI realizes this alternate approach by extending the well-known *publish-subscribe communication architecture* [10] to storage feature

HotStorage'22, July 27-28, 2022, virtual conference

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/1234.5678

development. As our first contribution, we discuss how Seagate's CORTX distributed storage system [42] can be extended to incorporate FDMI. FDMI allows new storage features, or *FDMI plugins*, to subscribe to and make consistent storage system changes in response to client-initiated operations. FDMI implements *transactional coupling*, whereby plugin operations are guaranteed to be executed atomically either simultaneously with or upon a successful completion of a storage client initiated operation. Most importantly, the FDMI plugin architecture is loosely coupled such that the plugin itself can be independently developed and deployed.

We conduct a study of how FDMI can be used to develop three different classes of plugins of varying complexity, requirements, and functionality. While the loosely coupled model may introduce additional I/O overhead in some instances, use of CORTX's enhanced client API for FDMI plugins can mitigate such effects. We believe that FDMI offers an intriguing new design point for building next-generation, scalable, extensible storage systems.

2 THE CASE FOR PUB-SUB STORAGE

Storage systems are often re-factored and designed to include new features that were not part of the original system [3, 11, 21, 30, 49]. This *Integrated* design requires a deep understanding of the codebase; the implementation is often cumbersome and error prone. To ease the development burden, previous works have proposed *Interposed I/O* that uses interposition to enable adding new features relatively independent of the storage system implementation itself [17, 50]. We refer to these feature implementations as *plugins*. The Conventional *Publish/Subscribe* (Conv. Pub-Sub) architecture allows dynamically attaching software components to large systems [5, 6, 13, 19, 24]. These Pub-Sub solutions focus on scaling the entire software service as opposed to augmenting the storage layer. In contrast, FDMI adopts the classic Pub-Sub communication architecture for storage systems but additionally augments the storage system with transactional support for subscriber operations.

Table 1 compares different properties across these plugin architectures including our proposed *FDMI* architecture. To allow for the widest possible range of plugins, the plugin architecture must support several key properties.

Access to the Storage layer. The ability to access the internal state and parameters of the storage layer is desirable. Integrated and Interposed architectures, by design, place plugins directly in the I/O path and can access client-issued I/O operations [17, 39, 50]. The Conventional Pub-Sub plugins are entirely outside the storage I/O path and do not have access to the storage layer. FDMI in contrast, is a storage-aware pub-sub architecture that has access to storage state and can optionally interpose itself in the I/O path.

Transactional guarantees. A key challenge with designing new storage plugins is allowing access and modifications without introducing inconsistency or corruption. An important distinction for FDMI is that it provides essential transactional guarantees to new storage plugins. FDMI plugins can perform storage operations within a transaction context provided by the storage core and thereby can achieve fault-tolerance and consistency. We discuss FDMI's *transactional coupling* design further in Section §3.

No I/O Amplification. Storage features may modify or generate storage I/O depending on the nature of their functionality. First, in case a plugin needs to access stored data for performing actions outside of the client I/O path, this will result in I/O amplification in all architectures – Integrated, Interposed, and FDMI. In case of operations triggered synchronously with client I/O, Integrated and Interposed plugins can access and modify the I/O operation payload and meta-data without needing additional storage-level operations. In contrast, FDMI allows an asynchronous operation in response to completed or ongoing I/O operations. Any additional I/Os to access the original client I/O payload leads to I/O amplification. However, there is potential for optimizations in the FDMI model that could offset some of this I/O amplification as discussed later in the paper (§6).

Independent Scaling. Depending on the complexity of the storage plugin, it may need greater or lesser resources than that of the storage system. Thus, allowing the plugin instance to scale independently is desirable but Integrated architectures are unable to achieve this goal. Interposed I/O, depending on the specific implementation, can allow for independent scaling of the plugin. For instance, FUSE-based plugins [1] *can* be deployed on independent resources with networked implementations, while other solutions at the block layer [17] are unable to do so. FDMI fundamentally decouples the storage system core layers from the storage plugin to achieve independent scaling.

Ease of development. The amount of developer effort is directly proportional to the development complexity of any software. The Integrated architecture imposes significantly higher complexity than other architectures since the developer must modify an existing complex codebase to implement the new feature. Interposed architectures can also incur modestly high complexity since plugin development often occurs in a highly concurrent environment and operations must be performed in the latency sensitive I/O path. FDMI and Pub-Sub architectures simplify the development process for plugin developers via a simple user level API.

3 FDMI OVERVIEW

In this section, we discuss how FDMI extends the core functionality of a storage system and enables the development

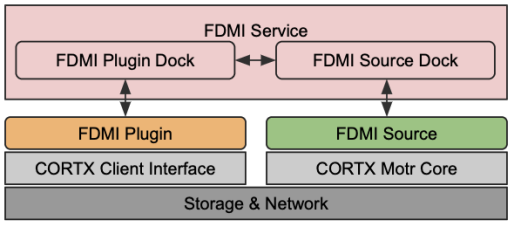

Figure 1: FDMI system architecture in CORTX.

of FDMI plugins. While we discuss FDMI's mechanisms in the context of Seagate's CORTX distributed storage system, FDMI's general design is more broadly applicable.

3.1 Seagate's CORTX

CORTX is Seagate's cloud software stack that provides distributed object storage [42]. The core storage component in CORTX is *Motr* [43]. Motr provides object and key-value storage and exposes the CORTX Client Interface and the CORTX Management and Control Interface. A CORTX deployment consists of several nodes connected over the network that run Motr server instances and other CORTX components. The software stack also includes additional layers to connect with NFS, CIFS, POSIX and S3/Swift protocols. There are other components for high availability checks and control such as HA, CORTX Provisioner, and CORTX Administration toolset.

3.2 Minimizing the Contact Surface

Besides decoupling the plugin development and deployment from the storage system, FDMI minimizes its own contact surface with the CORTX codebase. FDMI consists of three types of components, the *FDMI Source*, the *FDMI Service*, and *FDMI Plugin*. FDMI's contact surface is encapsulated entirely within its FDMI Source component which modifies CORTX's Motr core layer. The FDMI Service component is broken up into *FDMI Source Dock* and *FDMI Plugin Dock*; these are initialized with the Motr instance. The FDMI Plugin Dock interfaces with the FDMI Plugin instances while the FDMI Source Dock interacts with FDMI Source instances. Figure 1 depicts FDMI architecture within CORTX. The FDMI Source instances run as part of the Motr core storage layer. The FDMI plugin interacts with the FDMI plugin dock and can also optionally use the CORTX client interface if needed.

In a standard Motr cluster deployment, there may exist multiple FDMI Sources, one per Motr instance, each contributing a different set of information to the FDMI Source dock. The FDMI Source instance is integrated with Motr core and interposes on plugin-subscribed I/O operations, notifying the FDMI plugin via the FDMI service. In addition, it coordinates the committing of transactions at the Motr core in coordination with the FDMI plugin as necessary. Motr operations are initiated by clients accessing or updating the data or metadata of the storage system and are interposed upon by the FDMI Source. Each operation of interest to plugin and corresponding data that is sent through the system is an *FDMI record*. *FDMI filters*, maintained within the FDMI Source, define the set of rules to filter out FDMI records. Each filter rule has a cluster-wide, unique *Filter ID*.

3.3 A Narrow API

Applications running on different clients issue storage operations to the Motr system after it is up and initialized. These operations are of the plugin's interest and meet the filter rules defined in the cluster configuration service. We illustrate the interactions described below in Figure 2.

FDMI Source. The FDMI Source is part of a Motr server instance and is the only FDMI entity that manipulates CORTX state directly. When starting operation, the FDMI Source instances allocate an in-memory record structure for fast access during the time the record is processed by FDMI Source dock. A persistent version of this record is kept because multiple plugins may be in the process of receiving the records or running plugin operations. Records are discarded when all plugins have *released* the record.

FDMI Source Dock. The FDMI Source communicates with FDMI Source Dock using source specific record functions. It also updates reference counters associated with one record and informs the Source of FDMI record processing begin and end. The FDMI Source dock maintains a list of registered FDMI Sources with their corresponding posted records. The FDMI Source dock's main control flow runs a state machine that forwards notifications/responses to/from plugins.

FDMI Plugin Dock. The FDMI plugin dock interacts with each plugin and is responsible for registering and de-registering plugins and forwarding records to plugins. The plugin dock's private API allows plugins to access these functions and to enable/disable filters. The FDMI plugin dock also uses a state machine to deliver records to the plugin, update reference counters, and propagate messages to the Source.

FDMI Plugin. An FDMI plugin implements features we want to incorporate into the storage system. During initialization, plugins get added to the cluster as a *plugin-class* Motr client. Plugins then get access to the *extended* Motr client interface for issuing operations to Motr core. The extended Motr client interface is discussed further later (§4.2). Plugins are notified of FDMI records by the plugin dock. Every plugin explicitly indicates that a record is no longer needed by issuing a special *Release* message.

4 TRANSACTIONAL COUPLING

Storage features have a wide range of requirements for access to storage state as well as the ability to manipulate client I/O

operations. Before discussing how FDMI transforms CORTX into a pub-sub system supporting transactionally coupled plugins, we first identify and understand three classes of FDMI storage plugins.

4.1 A Taxonomy of Storage Plugins

We identify three classes of plugins that differ in how they interact with the storage system. In all cases, the plugin runs code in response to one or more client-initiated operations in the storage system.

Class A. Plugins in this class are simple consumers of client operations. They are notified of registered *operations of interest* after the containing "source" transaction has successfully *committed*. However, notifications must be reliable so that plugin actions can be made both fault-tolerant and transactionally consistent. An example of Class A plugin would be I/O profiling, whereby the plugin observes the I/O stream from the application.

Class B. Plugins in this class get notified similarly to Class A plugins, i.e., upon committing the precipitating transaction. However, Class B plugins generate additional operations back into the source storage system. The generated operations are *transactionally coupled* and are guaranteed to commit atomically if the plugin runs successfully. An example of a Class B plugin would be *semantic enhancer* plugin [33] that generates additional metadata of stored data asynchronously (e.g., automatic labeling of uploaded images).

Class C. Class C plugins are the most powerful. Plugins in this class get notified of operations *prior to* the source transaction commit. However, Class C plugins can augment and modify the source transaction itself with additional operations. The entire set of operations from both client and plugin get wrapped in a single transaction to be committed atomically. An example of a Class C plugin is a *dynamic tiering* plugin that changes data layout to different performance tiers of storage dynamically depending on data popularity.

4.2 Coupling Transactions

FDMI builds upon CORTX's distributed transaction support [14] to include a novel capability of *transactional coupling* for plugins.

The first variant of the transactional coupling model ensures that each plugin action will be coupled with the precipitating client transaction. Client transaction notifications are sent to the plugin, multiple times if needed to handle plugin failures. This guarantees that an active plugin stays consistent with the state of the storage system. The second variant allows plugins to reliably issue additional transactions in the storage system upon the success of the precipitating client transaction. Finally, the third variant allows plugins to augment and/or modify client-initiated operations with additional plugin operations, all of which are committed as part of a single joint transaction. This variant is the most powerful and allows the plugin to arbitrarily change how the storage system would respond to client I/O operations.

The foundation for FDMI's transactional coupling is *contextual* record logging whereby client operations, together with their transactional context, are persistently logged by the FDMI source. This means that, for every operation belonging to the same transaction, the corresponding FDMI record sent to the plugin inherits the precipitating transaction's context. For instance, consider two consecutive writes to different objects issued by different clients and both writes are part of the same transaction by CORTX's distributed transaction manager. For a Class A plugin that is attached and registered to monitor object writes, the source will have access to one FDMI event per committed operation. If the transaction is aborted by the client, then the corresponding FDMI plugin will not get notified and thus remain consistent with the storage system. Furthermore, a persistent log of committed transactions and registered plugins is used by the distributed transaction manager to handle the re-registration of plugins upon recovery after a failure.

Class B plugins in FDMI can additionally use the CORTX client interface to initiate and finalize new transactions. Plugins within this class use the second variant of the transactional coupling mechanism whereby they initiate new transactions that are dependent on the precipitating transaction committing. Each plugin-initiated transaction is a response to the corresponding FDMI received record(s) which is guaranteed to be part of a committed transaction. If the plugin fails after the precipitating transaction commits, operations within the newly created but unfinished plugin-initiated transaction can be restarted.

To support Class C plugins, the FDMI source forwards the precipitating event information along with the transactional context. To do so, Class C plugins use an *extended Motr client interface* that includes additional operations to *add*, *cancel* and *replace* operations within the same transaction context at the FDMI source. To make this possible, the source inserts itself into the precipitating transaction's context and request path before any response is sent back to the client. For instance, a Class C dynamic storage tiering plugin would remap client read and write operations depending on the dynamic location of data within performance tiers.

4.3 Life-cycle of an FDMI Record

To understand how FDMI's transactional coupling design works in its entirety, we can analyze the life cycle of an FDMI record for each plugin class. The Motr system is first initialized with a client application accessing storage. A set of filter rules are defined in the cluster configuration service

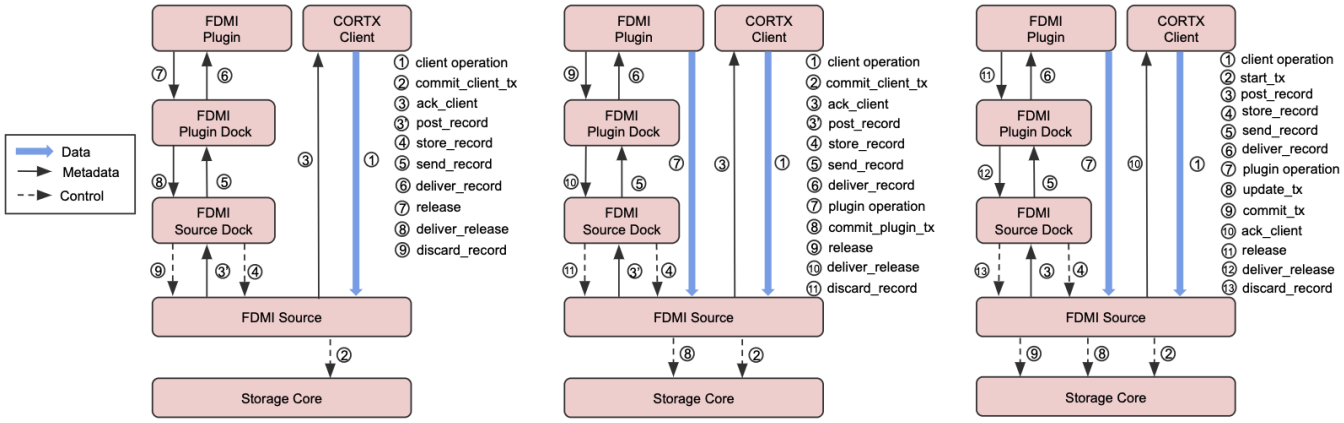

Figure 2: Life cycle of a single FDMI record for Class A (left), Class B (center) and Class C (right).

and FDMI record types are mapped to various plugins as they register themselves.

Figure 2 depicts the life-cycle of a single FDMI record produced at the source in response to a client operation and the plugin that registers its interest in the record. Three life-cycles one each for Class A (left), Class B (center) and Class C (right), are shown. In response to a client operation, for Class A and Class B plugins, the Source immediately commits the containing client transaction and acknowledges the completion of the operation to the client. Class C plugins, on the other hand, the Source simply starts a transaction in anticipation of additional plugin-initiated operations that will modify the transaction; client acknowledgment is delayed until after the transaction is committed at a later time.

As the next step, for each class, a record is posted by the FDMI Source to the Source Dock, thereby beginning its life-cycle. The Source Dock then starts analyzing the record filter rules and sends a message to the Source to inform that the processing has started and the record is then stored at the Source to be used in case of failures. After this process is done, Source Dock sends the record to the Plugin Dock which in turn notifies the plugin itself of the new record for plugin processing. In case of Class B and C, in response to the Plugin Dock's notification of the new record, the FDMI plugin acting as an FDMI client initiates its own transaction at the FDMI source directly. The FDMI source either commits (for Class B) or updates and then commits (for Class C) the plugin-initiated transaction and acknowledges the plugin. The Plugin Dock next forwards the plugin's *record release* notification to the Source Dock, which in turn asks the FDMI source to discard the record, thereby ending its life-cycle.

4.4 Source and Plugin Fault-Tolerance

FDMI provides fault tolerance for plugins and sources. A contextual record is guaranteed to be stored persistently within the FDMI source and available until all notified plugins explicitly release the record. Plugins that perform idempotent actions can recover from crashes simply by re-registering and getting notified of unreleased records. The FDMI Sources rely on CORTX's distributed transaction manager and its write-ahead log for crash recovery. The Source triggers a recovery phase for all active transactions and associated records. As a result, any registered plugin will receive all the records for further execution. Duplicate records, if received, are discarded by the plugin. Duplicate release operations as received by the FDMI Service are also discarded.

5 PRELIMINARY ANALYSIS

To understand the limitations and benefits of FDMI, we analyze a list of typical storage plugins. We evaluate plugin properties (Table 1) of each plugin and how each plugin architecture addresses them. Note that we did not consider the conventional Pub-Sub architecture since it only provides the ability to monitor the system from layers above the storage stack. Table 2 summarizes our findings.

Independent scaling and ease of development are achievable only through FDMI. Transactional guarantees are possible with Integrated plugins since the plugins have direct access to the transactional context and control flow within the storage layer. On the other hand, Interposed plugins are limited to the interface exposed by the storage system. Integrated and Interposed plugins eliminate I/O amplification while FDMI plugins that issue additional I/O to storage are unable to due to their asynchronous operation.

We also examined how various storage plugins map to FDMI classes. We first note that plugins that simply monitor activity within the storage system are Class A. Plugins such as *I/O offloading*, *I/O shepherding*, *dynamic tiering*, and *caching* are Class C plugins with inline operations requiring the direct modification of the I/O request stream. For

Storage Plugins	Integrated					Interposed					FDMI					Description
	S	T	N	E	D	S	T	N	E	D	S	T	N	E	D	
I/O Profiling [20]	■	■	■			■		■			■	A		■	■	Record the I/O stream per storage application.
System Profiling [2, 37]	■	■	■			■		■			■	A		■	■	Monitor internal parameters of storage system.
Backup/Replication [25, 36]	■	■	■			■		■			■	B,C		■	■	Replicate a consistent copy of the data.
Deduplication [27, 29]	■	■	■			■		■			■	B,C		■	■	Eliminate duplicated data to reduce space.
Encryption [12, 28]	■	■	■			■		■			■	B,C		■	■	Encrypt data for security purpose.
Compression [29, 51]	■	■	■			■		■			■	B,C		■	■	Compress data to reduce space utilization.
Integrity Checker [26, 34]	■	■	■			■		■			■	B,C		■	■	Detect inconsistencies and report errors.
RAID Mirroring [23, 35]	■	■	■			■		■			■	B,C		■	■	Create a copy of the data in different locations.
Semantic Enhancer [33]	■	■	■			■		■			■	B,C		■	■	Label data with semantic names.
Versioning [7, 40]	■	■	■			■		■			■	B,C		■	■	Maintain a backup of different versions.
Data Reorganization [9, 44]	■	■	■			■		■			■	B,C		■	■	Dynamically change data layout or striping.
Tiering [16, 46]	■	■	■			■		■			■	C		■	■	Optimize data placement among storage tiers.
Caching [41, 48]	■	■	■			■		■			■	C		■	■	Cache frequently accessed data.
I/O off-loading [32]	■	■	■			■		■			■	C		■	■	Redirect I/O traffic to another location.
I/O shepherding [18]	■	■	■			■		■			■	C		■	■	Handle I/O errors to improve data reliability.

Table 2: Storage plugins and how they can be implemented using the different system architectures. *Each property in the architecture column corresponds to one column on Table 1,* **S***: Storage Access,* **T***: Transactional Guarantees,* **N***: No I/O Amplification,* **E***: Independent Scale and* **D***: Ease of development. Darker cells indicate that specific property can be achieved using the corresponding architecture. The values within each cell for FDMI (column T) represent plugin class variants:* **A**, **B**, **C**.

example, a caching plugin may modify a client I/O operation to redirect it to the cache. On the other hand, plugins such as *backup/replication*, *deduplication encryption*, *compression*, *integrity checker*, *raid mirroring*, *semantic enhancer*, *versioning* and *data reorganization* require additional I/O for data/metadata updates and can be implemented as either Class B (asynchronous, offline) or C (inline) plugins.

6 DISCUSSION AND FUTURE WORK

The publish-subscribe model for storage is functionally feasible but it raises several questions around its scope of applicability, performance impact, and generalizability. An important next step is to further understand the potential and limitations of this new storage system architecture by implementing a wide range of plugins. In particular, since plugins vary in their consistency and performance requirements, evaluating how the broadest range of plugins can be supported within the publish-subscribe model is critical.

A key challenge with this evaluation is ensuring that plugins do not compromise storage performance guarantees. One avenue of exploration in this regard is further extending the client API to allow plugins to reduce the storage and network I/O amplification incurred. First, if recent client I/O payloads are cached at the FDMI source, the plugin initiated operations may avoid storage-level I/O amplification. Second, the FDMI API can be additionally augmented to offload some computation to the FDMI Source by allowing plugins to register functions that get executed in response to specific client I/O operations. Besides performing computation on data at the Source, these functions can also eliminate network and storage I/O amplification induced by the plugin.

Finally, we will also evaluate how the FDMI architecture applies to other storage systems such as Ceph [47], Swift [45], and Minio [31]. We believe the basic mechanisms of transaction logging and notification could be created within these alternate systems without significant development effort. The FDMI Service itself is independent of the store and only interacts with the FDMI Source and the plugin. Thus, we anticipate that building capabilities equivalent to the transactional coupling provided by the FDMI Source implementation in CORTX should be feasible within these alternate storage systems.

7 CONCLUSIONS

We presented the position that a publish-subscribe architecture is a valuable choice for building next generation storage systems. Our goals were multi-fold but the jumping board was not only the strong desire to enable storage system feature development in an environment free from the complexities of the storage system codebase but also the need to support the development of a wide range of features, safely. In our explorations, we realized the basic need for synchronizing the actions performed by the new feature – the subscribing plugin – with the actions of the storage system itself. The proposed transactional coupling mechanism enables the loosely coupled yet safe development of a wide range of plugins of different classes ranging from simple observers of storage activity to those that alter the functioning of the storage system itself. We expect future work in this space to further inform the utility and limitations of this new storage architecture.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We thank members of Seagate's CORTX team including Nikita Danilov, Hua Huang, Sai Narasimhamurthy, and Ganesan Umanesan, for sharing their knowledge of CORTX. Liana Valdes and Raju Rangaswami were supported in part by NSF award CNS-1956229, an ED GAANN fellowship, and a Seagate Technology research gift.

REFERENCES

[1] 2022. File Systems in the Linux kernel: FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[2] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2009. Generating Realistic Impressions for File-System Benchmarking. In *7th USENIX Symposium on File and Storage Technologies*.

[3] Marcos K. Aguilera, Kimberly Keeton, Arif Merchant, Kiran-Kumar Muniswamy-Reddy, and Mustafa Uysal. 2007. Improving Recoverability in Multi-tier Storage Systems. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 677–686.

[4] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. 2009. The Case for a Versatile Storage System. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'09)*.

[5] Amazon AWS. 2022. Amazon S3 Event Notifications. https://docs.aws.amazon.com/AmazonS3/latest/userguide/NotificationHowTo.html.

[6] Amazon AWS. 2022. AWS Lambda service. https://aws.amazon.com/lambda/.

[7] Amazon AWS. 2022. S3 Bucket Backup. https://docs.aws.amazon.com/aws-backup/latest/devguide/s3-backups.html.

[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.

[9] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, and Vagelis Hristidis. 2009. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*.

[10] K. Birman and T. Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. 123âĂŞ138.

[11] Matt Blaze. 1993. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS '93)*. 9âĂŞ16.

[12] Giuseppe Cattaneo, Luigi Catuogno:Università di Salerno, Aniello Del Sorbo:Università di Salerno, and Pino Persiano:Università di Salerno. 2001. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*.

[13] Ceph. 2022. Bucket Notifications. https://docs.ceph.com/en/latest/radosgw/notifications/.

[14] Nikita Yurievich Danilov and Eric Barton. 2012. System and method for performing distributed transactions using global epochs. US Patent 8,103,643.

[15] Michail Flouris and Angelos Bilas. 2005. Violin: A Framework For Extensible Block-level Storage. In *IEEE Conference on Mass Storage Systems and Technologies*.

[16] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage Using Extent Based Dynamic Tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. 20.

[17] Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami. 2008. The case for active block layer extensions. *ACM SIGOPS Operating Systems Review* 42 (10 2008), 3–9.

[18] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving File System Reliability with I/O Shepherding. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. New York, NY, USA, 293âĂŞ306.

[19] Hadoop. 2022. Zookeeper Watches. https://zookeeper.apache.org/doc/r3.3.3/zookeeperProgrammers.html.

[20] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *ACM Trans. Comput. Syst.* (aug 2012), 39.

[21] Dave Hitz, Michael Malcolm, and James Lau. 1994. File System Design for an NFS File Server Appliance. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*.

[22] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. 2005. The Automatic Improvement of Locality in Storage Systems. *ACM Trans. Comput. Syst.* (nov 2005), 424âĂŞ473.

[23] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 355–370.

[24] Kafka. 2022. Kafka Documentation. https://kafka.apache.org/documentation/.

[25] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. 2004. Designing for Disasters. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*.

[26] Gene H. Kim and Eugene H. Spafford. 1994. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS '94)*. 18âĂŞ29.

[27] Ricardo Koller and Raju Rangaswami. 2010. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*.

[28] Andrew W. Leung, Ethan L. Miller, and Stephanie Jones. 2007. Scalable security for petascale parallel file systems. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[29] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*.

[30] Linux. 2022. LessFS: deduplication filse system in Linux. https://sites.google.com/a/projectme.org/lessfs/lessfs-guide.

[31] MinIO. 2022. MinIO: Multi-Cloud Object Storage. https://min.io.

[32] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write Off-Loading: Practical Power Management for Enterprise Storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*.

[33] James O'Toole, David Gifford, Pierre Jouvelot, and Mark Sheldon. 1997. Semantic File Systems. *ACM SIGOPS Operating Systems Review* 25 (11 1997).

[34] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. 2004. FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Conference on System Administration (LISA '04)*. 67âĂŞ78.

[35] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. 109âĂŞ116.

[36] R. Hugo Patterson and Stephen Manley. 2002. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Conference on File and Storage Technologies (FAST 02)*.

[37] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. 8.

[38] Tom Rhodes. 2007. FreeBSD Handbook - Chapter 19: GEOM: Modular Disk Transformation Framework. *FreeBSD Handbook* (2007).

[39] David Rosenthal. 1990. Evolving the Vnode Interface. In *In USENIX Conference Proceedings*. 107–118.

[40] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. 1999. Deciding When to Forget in the Elephant File System. *SIGOPS Oper. Syst. Rev.* (dec 1999), 110âĂŞ123.

[41] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. 267âĂŞ280.

[42] Seagate. 2022. CORTX Intelligent Object Storage Software. https://www.seagate.com/products/storage/object-storage-software/.

[43] Seagate. 2022. CORTX Motr. https://github.com/Seagate/cortx-motr.

[44] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2003. Semantically-Smart Disk Systems. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*.

[45] Swift. 2022. Swift: OpenStack Object Storage. https://wiki.openstack.org/wiki/Swift.

[46] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. 2010. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*.

[47] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. *Proc. USENIX OSDI* (November 2006).

[48] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 307–323.

[49] XiaoJian Wu and A. L. Narasimha Reddy. 2012. A Novel Approach to Manage A Hybrid Storage System. *J. Commun.* 7 (2012), 473–483.

[50] Erez Zadok, Ion Badulescu, and Alex Shender. 1999. Extending File Systems Using Stackable Templates. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. USENIX Association.

[51] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 121–128.