PathFS: A File System for the Hierarchical Edge

Vinicius Dantas de Lima Melo
University of Toronto

Myles Thiessen
University of Toronto

Aleksey Panas
University of Toronto

Alexandre da Silva Veith
Nokia Bell Labs

Keijiro Yano
Konica Minolta

Oana Balmau
McGill University

Eyal de Lara
University of Toronto

ABSTRACT

As IoT devices multiply and produce vast volumes of data, there is a heightened demand for instantaneous data processing. However, traditional cloud computing cannot adequately address these demands due to its latency and bandwidth limitations. Edge computing has emerged as a viable alternative with a hierarchical deployment of datacenters. However, this introduces additional layers of infrastructure and management that increase application development complexity. Using a shared file system is an attractive method for enhancing communication between components in an edge computing application.

In this paper we introduce PathFS, a shared file system designed for the hierarchical edge-cloud infrastructure. PathFS adopts a tree-like structure, with cloud datacenters at the root, edge datacenters as leaves, and a variable number of network datacenters in between. We evaluate PathFS through benchmarks on an emulated hierarchical edge deployment and compare it with NFS and ownCloud. The results show that PathFS offers lower latency than these systems by an order of magnitude, and scales to a larger number of concurrent clients without performance impacts, providing an end-to-end latency reduction of at least 80%.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Mobile Edge Computing, Cloud Computing, File Systems

ACM Reference Format:
Vinicius Dantas de Lima Melo, Myles Thiessen, Aleksey Panas, Alexandre da Silva Veith, Keijiro Yano, Oana Balmau, and Eyal de Lara. 2024. PathFS: A File System for the Hierarchical Edge. In *7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3642968.3654822

1 INTRODUCTION

Interconnected Internet of Things devices, including sensors, smartphones, and cameras are expected to generate 79.4 Zettabytes of data in 2025 [21]. This surge in data generation is accompanied by a growing demand for data processing. Relying solely on cloud computing for these tasks is inadequate due to issues of latency and bandwidth utilization. In response, edge computing has emerged as a viable alternative. It involves deploying a hierarchical network of datacenters between the wide-area cloud and end-user devices, enabling the exploitation of geographic proximity for efficient computation and storage access [28].

The hierarchical deployment of datacenters introduces additional layers of infrastructure and management that increase application development complexity. A shared file system presents an appealing approach to facilitate interaction among components within an edge computing application. The file system API is present in nearly all modern operating systems and devices, providing a standardized set of functions and protocols for managing files and directories.

Unfortunately, existing distributed file systems, such as NFS [22], AFS [10], Coda [12], HDFS [26], BlueFS [19], ownCloud [5], Hydra [29], and IPFS [4] are not well-suited for hierarchical edge deployments. First, many of these systems are primarily designed for datacenters or local area networks, where low latency and high bandwidth are prevalent. In contrast, edge networks experience higher latency and limited bandwidth, making traditional distributed systems inefficient in this context. Second, these systems assume a flat topology or homogenous servers. Edge deployments, however, are hierarchical and available resources gradually decrease as we descend the hierarchy. Third, several of these systems are based on peer-to-peer approaches that require data to be accessed across the wide area (e.g., IPFS), which is not compatible with edge applications that require low latency. Finally, systems such as Coda and ownCloud support clients with limited (and intermittent) connectivity; however, these systems assume a single layer of strongly connected servers, which is not realistic for edge deployment given the large number of datacenters involved and their placement.

We introduce PathFS, a new distributed file system designed for hierarchical edge deployments, such as the one presented in Figure 1. PathFS adopts a tree-like structure, with cloud datacenters at the root, edge datacenters as leaves, and a variable number of network datacenters in between. PathFS ensures low-latency data access by enabling *efficient local reads and writes.* Moreover, PathFS addresses the limited storage capacity in lower hierarchy levels by employing *on-demand partial replication*; files are persisted at the root of the hierarchy (i.e., the cloud) with additional layers acting

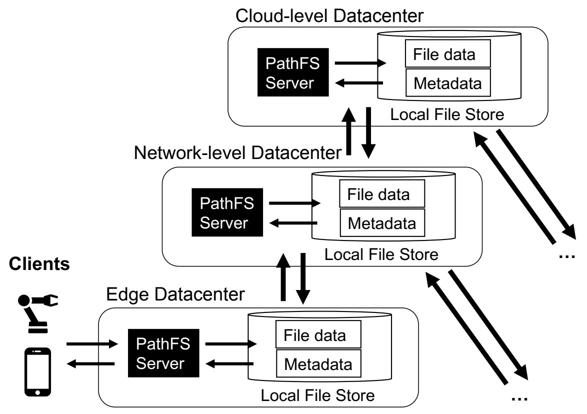

Figure 1: An example of PathFS' hierarchical topology.

as temporary caches. Our system supports a large replication factor via an eventual consistency model and leveraging the hierarchical nature of the deployment to minimize replica-tracking overhead. Finally, PathFS enables efficient communication among distributed processes through an event notification mechanism, eliminating polling, and enhancing overall system efficiency.

We implemented PathFS using Pathstore [18], a datastore for hierarchical edge deployments. PathFS provides a file system interface based on FUSE integration. Moreover, PathFS implements a distributed event notification mechanism using the inotify API [3] enabling efficient coordination between geo-distributed applications. We evaluated PathFS in a simulated hierarchical edge setup in multiple AWS datacenters. We compared PathFS with NFS [20] and ownCloud [5], an open-source clone of Dropbox. Our evaluation shows that PathFS' hierarchical design coupled with its event notification system, effectively minimizes end-to-end delays by localizing communication and negating the need for polling. Compared to NFS or ownCloud, PathFS reduces the latency between sensors and monitoring tasks by at least 80%.

In summary, this paper makes the following contributions:

- Identifies a list of requirements that a file system for the edge needs to satisfy. (Section 3)
- Presents PathFS, the first distributed file system which (1) leverages the multi-layer topology of hierarchical edge deployments and (2) provides support for distributed inotify notifications a feature which significantly reduces latency and bandwidth consumption. (Section 4)
- An implementation and evaluation of three use cases that experimentally quantifies the benefit of having access to a hierarchical file system. (Section 5)

2 RELATED WORK

There are two widespread ways to provide distributed file access: Clustered File Systems (CFS), and Distributed File Systems. In CFS, the data is distributed on multiple machines allowing multiple clients to gain direct disk access at the block level. Such systems expose block devices via a storage area network protocol like iSCSI [23], NVMe Over Fabrics [15], or fiber channels. Examples include GPFS [24], Lustre [7, 25], PVFS [8], and FaSST [11]. The main difference between PathFS and CFS systems lies in the assumptions made on the deployment scenario. While CFS assume a datacenter or high-performance computing cluster environment, PathFS was designed to leverage the topology of the hierarchical edge.

NFS [22] is one of the earliest distributed file systems, proposing a stateless file protocol that provides fast server crash recovery. Updated versions of NFS (e.g., NFS v4 [20]) are widely used in the industry (e.g., Amazon, IBM, Microsoft, Oracle). The Andrew File System (AFS) [10] and CODA [12] are other early distributed file systems that provide an alternative design to NFS. Overall, this design has had lower adoption than NFS. One of their key ideas is whole-file caching on the local disk of the client machine (as opposed to block-based caching), enabling fast subsequent reads and writes. While this stateful caching mechanism is similar to the on-demand replication in PathFS, these systems do not take advantage of the multi-layer hierarchical edge, assuming a client-server model. Moreover, the systems' consistency model does not allow for efficient streaming of continuous appends which is a common workload in edge computing, which is optimized by PathFS.

Many filesystems and datastores have been specifically tailored for edge deployments. The diversity of these solutions reflects the many competing design goals they target. BlueFS [19] is a file system designed for mobile devices that follows a client-server architecture and considers energy cost when deciding which device it will read from. ElfStore [16] and DynPubSub [6] use a peer-to-peer model for device and data management. Fogstore [9] and PathStore [18] are storage systems built on top of the Cassandra key-value store [13]. Fogstore provides differential consistency guarantees for clients, based on their context. PathStore is structured as a hierarchy of independent object stores where the root object store holds the entire dataset, while the other levels of the hierarchy hold partial data replicas. We leverage concepts from PathStore in the design of PathFS. However, substantial changes are required to support file system semantics on top of a datacenter hierarchy, including support for a distributed notification mechanism, conflict detection and resolution, and FUSE integration.

3 EDGE FILE SYSTEM REQUIREMENTS

A shared file system for the edge needs to simultaneously satisfy the following requirements:

R1: Fast reads and writes. Real-time applications running on the edge demand low latency to ensure uninterrupted operation. For instance, real-time analytics and machine automation heavily rely on immediate access to data for instantaneous decision-making and responsiveness. This is especially crucial in time-sensitive scenarios where even a slight delay can result in significant consequences, such as in autonomous vehicles, and augmented reality applications.

R2: Local files. In edge applications, local files are essential not only for efficient data storage and immediate processing but also for conserving bandwidth. By enabling data to be stored and processed locally at an edge datacenter, the amount of data that needs to be transmitted to a centralized server or cloud infrastructure is significantly reduced resulting in substantial bandwidth savings.

R3: Effecient update notifications. Ably keeping track of changes is essential for applications that rely on the file system for communication and collaboration.

R4: Scale to a large number of datacenters. The limited coverage of edge computing sites requires high replication of data and services to ensure widespread availability and reliable performance.
R5: Work with limited storage capacity. As we approach the edge of the network, the available resources in a given datacenter decrease, including its storage capacity. While the cloud can accommodate the full file system, storing it in edge datacenters becomes impractical due to resource limitations.
R6: Support for intermittent connectivity. In edge computing, intermittent connectivity is common due to factors like signal interference, network congestion, or the mobility of edge devices. To ensure seamless operation, a file system must support offline operation and facilitate data recovery after network failures.
R7: Support for client mobility. For a consistent user experience across edge locations, an edge computing file system needs to offer session consistency, ensuring two key features: read-your-writes, and monotonic reads, meaning once a client has viewed a value for an object, future reads will either reflect that value or a newer one.

4 DESIGN AND IMPLEMENTATION

PathFS is structured as a hierarchy of nodes, where each node is run in a different datacenter consisting of one or more servers. Figure 1 shows a sample, three-layer PathFS deployment. The PathFS node at the root of the hierarchy is assumed to be persistent, while all other levels act as ephemeral partial replicas. This approach makes it possible to accommodate storage constraints as we get closer to the edge of the network (R5). To provide low latency (R1), files are replicated on demand so that read and write operations are performed against the PathFS node nearest to a client application. PathFS supports concurrent reads and writes on all nodes of the file system hierarchy; updates are propagated through the hierarchy in the background, providing eventual consistency.

Figure 2 shows the PathFS architecture. A node consists of a *local file store* and the PathFS *server*. The local file store provides persistent storage for objects that are temporarily replicated at a node. In our prototype, we use Cassandra [13] as a file store. The PathFS server handles requests from local clients, tracks file updates, handles notifications, and copies data between its local file store and the file store instance of its parent node. The PathFS client, consists of the FUSE [27] and inotify wrappers, and the PathFS driver. The client application interacts with PathFS through the FUSE and inotify wrappers, which call on the PathFS driver.

PathFS keeps track of metadata and data separately. The PathFS data schema is divided into two tables, one for the metadata, and one for file data. The metadata table stores the metadata of all entries in the file system (ie, files, directories, and symbolic links). Each entry in the Metadata table holds a reference to the UUID of its parent directory. We assume that the UUID belonging to the root directory (ie, /) is known. The file data table stores the data that belongs to each file, divided into equal *blocks*. Cold file entries are deprecated periodically, preventing the continuous fetching of updates for files that are no longer in use. In case of resource contention, we use a simple LRU policy to free space. To enable efficient local processing (R2), PathFS also provides local files for temporary storage. Updates to local files are not propagated to other nodes. The PathFS driver also integrates in-memory caches for metadata and file blocks.

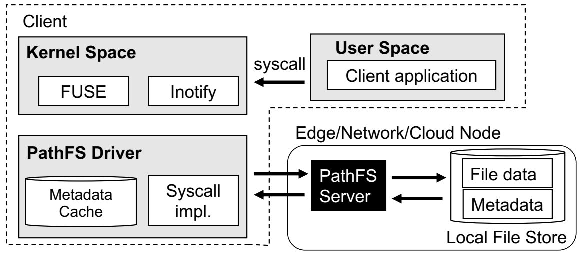

Figure 2: PathFS architecture.

4.1 FUSE Integration

FUSE integration makes it possible to mount PathFS on a local directory on the client like a standard file system. The FUSE kernel module forwards system calls to the PathFS driver, which transforms them into appropriate database queries that are sent to the PathFS server for execution. For instance, a `mkdir` command on the terminal invokes a series of system calls, such as `getattr`, `open`, and eventually `mkdir`. The PathFS driver provides the implementation of these system calls. For example, `mkdir` will execute an Insert query into the Metadata table with appropriate values.

To optimize performance, the PathFS driver integrates in-memory caches for both metadata and file blocks. The metadata cache is populated during path resolution. Similarly, the file-block cache retains a user-configurable number of blocks for files that are presently open. The file-block cache effectively manages the disparity in transfer sizes between Cassandra and FUSE. Cassandra is designed to excel in handling large data transfers, such as our deployment's optimal block size of 128KB. Conversely, FUSE operates by breaking accesses into smaller 4KB chunks. This incongruity is resolved by the file-block cache, ensuring that these smaller reads and writes are served directly from the cache without needing communication with the PathFS server. A dirty file block is written to the PathFS server (and Cassandra) when the cache gets full or when an application executes a flush operation or closes the file.

4.2 On-Demand Partial Replication

To maximize the use of limited storage capacity (R5), PathFS *replicates data on demand* when an application requests to open a file. This replication occurs at the level of individual files. When a file is opened for reading or updating, the PathFS server first checks if the file exists locally. If it is not present locally, the server requests said file from its parent node. If the file is not found at the parent node, the request is recursively forwarded to the next ancestor until it reaches the highest level of the hierarchy. Throughout this process, all metadata and file data blocks associated with said file are stored at each node along the path. When a file is opened for writing, the replication procedure is skipped, and the file is created locally without the need for traversing the hierarchy. Once the file operation is completed, all subsequent reading and writing actions on the file are handled by the local PathFS instance. The drawback of retrieving data exclusively upon request from applications is the noticeable delay caused by retrieving data across multiple levels of the hierarchy. It is easy to envision alternative approaches that proactively fetch data in anticipation of its future usage.

4.3 Consistency Model and Update Propagation

To enable a high degree of georeplication (R4), PathFS provides optimistic replication with eventual consistency. All modifications are applied locally, and a background process periodically propagates local updates to the node's parent, which in turn forwards them up the hierarchy until they reach the root node. Similarly, each node also periodically fetches updates from its parent node.

PathFS uses a write log to store changes to file data by tagging rows in the metadata and filedata tables with a globally unique identifier. Each identifier includes a physical timestamp and the issuing node's unique identifier. As modifications propagate throughout the hierarchy, PathFS uses these identifiers to order modifications. Specifically, modifications are first ordered by their physical timestamp and ties are broken by their node identifier. To ensure this ordering captures temporal order, all nodes in PathFS are required to synchronize their physical clocks with NTP [14].

PathFS' write log is not visible to applications. Instead, the PathFS server automatically collects multiple versions of a row and returns the most recent data. In the current prototype, conflicts are resolved by a simple policy, in which the modification with the most recent timestamp wins. Since all nodes are equipped with synchronized clocks, this policy approximately provides last-writer wins semantics, a standard concurrency model for file systems.

4.4 Update Notifications with Inotify

To make it possible for geo-distributed applications to use the file system for coordination (R3), PathFS lets applications register to receive notifications of file system updates. PathFS' notification mechanism extends the inotify monitoring API, which only supports local file systems. Applications can set watches on directories (add/delete file) or files. An event is triggered when a watched file or directory gets updated as a result of a local write or data propagation through the hierarchy. To tolerate message loss, events are periodically re-transmitted at most a configurable number of times or until the server receives an acknowledgement from the client.

Applications need to add a watch for each monitored file or directory. The watch returns a file descriptor through which the user can read all incoming events. An event is a small structure containing the event type (e.g., new file, change in file attributes, etc.) and the absolute path of the affected file or directory. Once PathFS receives the events, it forwards them to all watches that are established on the corresponding file or directory. Events are buffered by PathFS, so when the overridden inotify library calls `read`, it returns the first unsent event from the buffer.

4.5 Fault Tolerance

PathFS can continue to serve reads for files that are locally replicated even in the event of a network partition (R6); however, accesses to files that are not already in the local store of the current node (and its reachable ancestors) will fail if an ancestor becomes unreachable. On the other hand, writes can be executed as long as the local PathFS instance is reachable. A write returns when it is persisted in the local file store instance, and it is guaranteed to remain stored in the local instance until it is propagated to the parent node. A file block is marked *dirty* when it is inserted into the local file store instance. PathFS only marks the block as *clean* when the parent acknowledges reception and storage of the write. If there is a failure, PathFS retries propagating the write. If a PathFS node experiences a temporary failure, upon recovery it will retry propagating all writes locally marked as dirty.

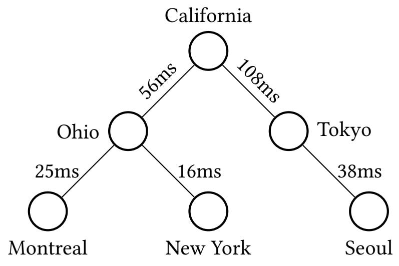

Figure 3: Experimental topology with round-trip latencies.

4.6 Client Mobility

PathFS supports client mobility (R7) by enforcing session consistency when a client transitions to a different node. This ensures that a client will always read their most recent writes, meaning after an update the client's ensuing reads will display either the updated data or a more updated version. Furthermore, monotonic reads are maintained, which means that once a client sees a value for an object, subsequent reads will present that value or a newer one. The method PathFS adopts to maintain session consistency mirrors the approach utilized by SessionStore [17]. We cluster relevant file system operations into sessions and monitor all the files accessed or modified during a session. When a client shifts between PathFS nodes, we verify that the same (or newer) versions of the files related to their session are available on the new node before performing any file system operations.

5 EVALUATION

We conducted our evaluation on an emulated hierarchical edge network consisting of virtual machines deployed on different AWS datacenters. The hierarchical topology is shown in Figure 3. The links are labeled with the measured average round-trip latencies. California is the cloud (root) datacenter, Ohio and Tokyo host the network-level (core) datacenters, and Montreal, New York, and Seoul are the edge-level datacenters. The available bandwidth between datacenters is up to 10 Gbps. The client applications run on separate EC2 instances. The PathFS block size is set to 128KB. We compare PathFS against NFS v4.2 [20] and ownCloud [5], an open-source clone of Dropbox. The NFS and ownCloud servers are deployed at the California datacenter. To experimentally quantify the benefit of PathFS we consider three scenarios:

1. **Software distribution** measures the time to deploy a new software version. This scenario highlights the benefits of partial replication and data locality.
2. **Object detection** is a common IoT use case where multiple cameras capture images and send them for analysis on a remote datacenter illustrating the importance of remote notifications.
3. **Temperature monitoring** is another IoT use case where captured data is small and periodic that showcases the need for efficient streaming of continuous appends.

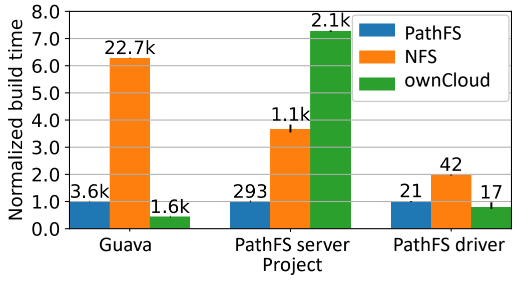

Figure 4: Normalized software distribution build times. Absolute build times (in seconds) shown on top the bars.

5.1 Software Distribution

We used the edge file system to deploy a new software version to edge devices. We built three Java projects: (1) Guava library release 32.0.1[2] (3197 Java files); (2) PathFS server codebase (130 Java files); and (3) PathFS driver that includes 8 Java classes and approximately 3,000 lines of code in total. The scenario assumes that the *developer* who creates the new software is located in California and the *deployer* who deploys the new version on their device is in Montreal (two hops away with an average total RTT of 81 milliseconds). The experiment starts with the developer copying the new version of the software to a shared directory on the California datacenter. Then, in Montreal, the deployer accesses the shared directory and builds using Maven.

Figure 4 shows the average time it takes to build Guava, the PathFS server, and the PathFS driver over PathFS, NFS, and ownCloud. Results are averages over ten runs and are normalized by the PathFS values, the absolute numbers in seconds are shown atop the bars. Standard deviations are below 10% in all cases.

NFS takes longer than PathFS on all projects. Both PathFS and NFS use an on-demand replication approach that ensures that unnecessary files, such as git history or other modules or libraries that are present in the repository but are not accessed as part of the build process are not fetched. However, NFS does not fully leverage data locality, as it does not have permanent storage local to edge or core datacenters. For example, all writes require blocking communication with the root datacenter. PathFS performs them to the local node's storage and propagates them in the background. The absence of data locality, especially on big projects, made NFS at least 2 times slower than PathFS.

ownCloud requires a full synchronization that pulls all files from the server. Every sync request runs a lookup on all files and directories to detect changes since the last sync, updated files are then pulled. This policy is beneficial when most of the files will be used by the build process (the Guava and PathFS driver cases). However, this approach can lead to slow build times if there are unused files, such as other modules or libraries present in the same repository, or a large volume of documentation files. This is the case for PathFS server codebase, resulting in the building process taking 7 times longer with ownCloud compared to PathFS.

5.2 Object Detection

We emulate a set of cameras regularly capturing images of a production line. The images are then processed by an image analysis job that performs object detection. The cameras are connected to the New York edge node, and the object detection task runs on the Ohio core node. Every 30 seconds, each camera copies a 200KB image from the COCO dataset [1] to the shared directory as a new file. We vary the number of cameras between 1 and 32. Cameras start with random delays to ensure that the workload is evenly spread. For PathFS, the object detection job uses inotify to find out if a new image was created. For NFS and ownCloud, which do not provide a notification mechanism, the object detection job polls the directory to check if new files have been added.

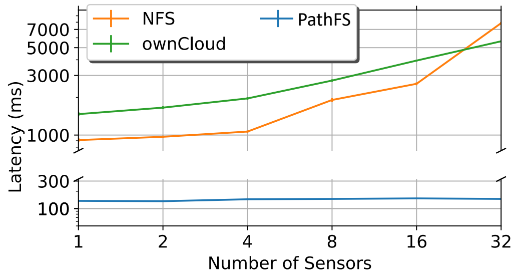

Figure 5: Object detection average latency.

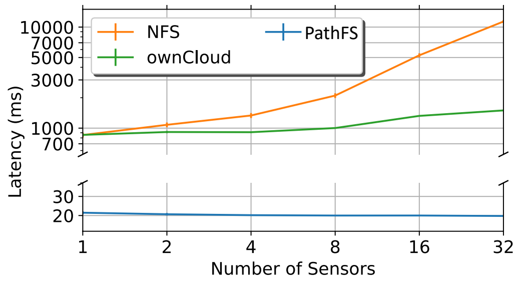

Figure 6: Temperature monitoring average latency.

Figure 5 shows the end-to-end latency, measured from when the camera starts writing a new image until the object detection task finishes the analytics job on that image. PathFS is 85% and 90% faster than NFS and ownCloud, respectively. PathFS also shows better scalability, with only a slight increase in average latency as the number of clients varies from 1 to 32. In contrast, NFS exhibits rapid degradation in performance as the number of clients increases. While ownCloud performs better than NFS, it also experiences a rise in average latency when the number of clients exceeds 8. The advantages of PathFS are due to two reasons. First, its hierarchical design enables direct transmission of data from New York to Ohio. Second, PathFS benefits from update notifications. The object detection job establishes a watch on `/$mountpoint/images`. In contrast, NFS and ownCloud require all requests to pass through the server running in California, and the lack of a notification mechanism requires numerous polling attempts.

5.3 Temperature Monitoring

This IoT scenario emulates a collection of thermometers connected to the New York datacenter that capture periodic readings. Each thermometer appends its temperature reading every 40 seconds to

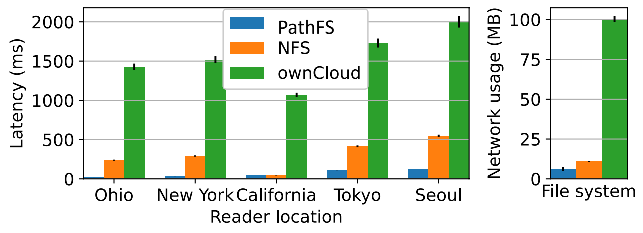

Figure 7: PathFS's event notification mechanism lowers overall latency and bandwidth compared to polling.

a separate existing file, resulting in writes of 3 bytes. A monitoring job running in Ohio calculates the global average temperature for a given sensor whenever a new reading is added. The experiment is conducted for 45 minutes. End-to-end latency is defined as the time elapsed between a thermometer appending a new reading and the monitoring task recalculating the new average value. Figure 6 shows that PathFS' latency is substantially lower than the Object Detection scenario (20 ms vs. 140 ms) as the writes are smaller. NFS, on the other hand, becomes at least 40 times slower than PathFS because of the constant metadata and content changes.

5.4 Polling vs. Notifications Microbenchmark

Finally, we assess the benefits of remote notifications. For this purpose, we place a single writer in Montreal that creates a new directory every 10 seconds. We then start 6 readers on all other nodes in the topology. We measure the time it takes for each reader to notice the creation of the new directory. Figure 7 shows the average time it takes for readers located on different parts of the topology to learn about the creation of the new directory. As expected, PathFS' notification mechanism makes it possible for readers to recognize new directories in a fraction of the time compared to NFS and ownCloud. The exception is California where NFS achieves better performance because the NFS server is located on this node. Additionally, we show the total bandwidth consumed under PathFS, NFS, and ownCloud. Not surprisingly, PathFS requires much less network bandwidth.

6 CONCLUSION AND FUTURE WORK

PathFS is a distributed file system designed for hierarchical edge deployments, embracing their inherent tree-like topology. PathFS ensures low-latency data access through efficient local reads and writes. PathFS significantly reduces end-to-end latency in sensor-to-monitoring jobs compared to NFS and ownCloud. There are several directions we plan to explore in future work on PathFS. First, PathFS has the potential to utilize its detailed understanding of the data previously accessed by applications to predict its future usage patterns. The exploration of different prefetching methods is an area for future investigation. Second, though our client mobility strategy is straightforward, it could result in waiting times since the client may need to pause to allow the data to traverse the PathFS hierarchy. Delving into mechanisms that can foresee client movement and preemptively replicate data is another area for future exploration. Finally, we intend to explore alternative conflict resolution approaches as part of our future work.

REFERENCES

[1] [n. d.]. Common Objects in Context (COCO). https://cocodataset.org.
[2] [n. d.]. Guava. https://github.com/google/guava/releases/tag/v32.0.1.
[3] [n. d.]. inotify. https://man7.org/linux/man-pages/man7/inotify.7.html.
[4] [n. d.]. InterPlanetary File System (IPFS). https://ipfs.tech/.
[5] [n. d.]. ownCloud. https://owncloud.com/.
[6] Chamseddine Bouallegue and Julien Gascon-Samson. 2020. DynPubSub: A Peer-to-peer Overlay for Topic-based Pub/sub Systems Deployed at the Edge. In *Proceedings of the International Middleware Conference Demos and Posters.*
[7] Peter Braam. 2019. The Lustre Storage Architecture. *arXiv preprint arXiv:1903.01955* (2019).
[8] Philip H Carns, Walter B Ligon III, Robert B Ross, and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the Annual Linux Showcase Conference.*
[9] Harshit Gupta and Umakishore Ramachandran. 2018. Fogstore: A Geo-distributed Key-value Store Guaranteeing Low Latency for Strongly Consistent Access. In *Proceedings of the ACM International Conference on Distributed and Event-based Systems.*
[10] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. 1987. Scale and Performance in a Distributed File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP).*
[11] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*
[12] James J. Kistler and M. Satyanarayanan. 1991. Disconnected Operation in the Coda File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP).*
[13] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review* (2010).
[14] David L Mills. 1985. *Network time protocol (NTP).* Technical Report.
[15] Dave Minturn and J Metz. 2015. Under the Hood with NVMe over Fabrics. In *Ethernet Storage Forum. SNIA.*
[16] Sumit Kumar Monga, Sheshadri K Ramachandra, and Yogesh Simmhan. 2019. ElfStore: A Resilient Data Storage Service for Federated Edge and Fog Resources. In *IEEE International Conference on Web Services (ICWS).*
[17] Seyed Hossein Mortazavi, Mohammad Salehe, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan PuzhavakathNarayanan. 2020. SessionStore: A Session-aware Datastore for the Edge. In *IEEE International Conference on Fog and Edge Computing (ICFEC).*
[18] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal De Lara. 2017. CloudPath: A Multi-Tier Cloud Computing Framework. In *ACM/IEEE Symposium on Edge Computing (SEC).*
[19] Edmund B Nightingale and Jason Flinn. 2004. Energy-efficiency and Storage Flexibility in the Blue File System.. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*
[20] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. 2000. The NFS Version 4 Protocol. In *Proceedings of the International System Administration and Networking Conference.*
[21] John Rydning, Marcia Walker, and Amy Machado. 2022. Worldwide IDC Global DataSphere IoT Device Installed Base and Data Generated Forecast, 2022-2026. In *International Data Corporation Market Forecast.*
[22] Russel Sandberg. 1986. The Sun Network File System: Design, Implementation and Experience. In *USENIX Technical Conference and Exhibition.*
[23] Julian Satran, Kalman Meth, C Sapuntzakis, M Chadalapaka, and E Zeidner. 2004. *Internet Small Computer Systems Interface (iSCSI).* Technical Report.
[24] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST).*
[25] Philip Schwan. 2003. Lustre: Building a File System for 1000-Node Clusters. In *Proceedings of the Linux Symposium.*
[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST).*
[27] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST).*
[28] Blesson Varghese, Eyal De Lara, Aaron Yi Ding, Cheol-Ho Hong, Flavio Bonomi, Schahram Dustdar, Paul Harvey, Peter Hewkin, Weisong Shi, Mark Thiele, et al. 2021. Revisiting the Arguments for Edge Computing Research. *IEEE Internet Computing* (2021).
[29] Shengan Zheng, Jingyu Wang, Dongliang Xue, Jiwu Shu, and Linpeng Huang. 2022. Hydra: A Decentralized File System for Persistent Memory and RDMA Networks. *IEEE Transactions on Parallel and Distributed Systems* (2022).