PathFS: A Streaming File System for Edge Environments

Kianoosh Abbasi
University of Toronto
Toronto, ON, Canada
kianoosh@cs.toronto.edu

ABSTRACT

As computers ubiquitously spread out globally, the current state-of-the-art cloud computing applications will hardly meet the demands in the near future. With the emergence of the internet of things (IoT), hundreds of smart devices will be connected in a small area, and merely putting the cloud servers in charge of the entire computation tasks will consume tremendous bandwidth and incur unacceptable latency. Edge computing presents a unique solution that exploits the computing power of each node and saves bandwidth considerably. That is, breaking down the computation tasks and executing them on various end-user devices (called edge devices), network cells or routers (called cloudlets), and the cloud servers. One motivating usage of edge computing is various always-on sensors and cameras that operate within a company, facility, or city. Human operators may want to access the live data collected by the sensors, and more powerful cloud servers may be running analytical algorithms on them. Therefore, a streaming file system tailored for edge computing environments can provide accessibility, scalability, low latency, and low bandwidth consumption. To achieve that goal, we build our desired system on top of PathStore, an edge computing storage system that offers a lightweight hierarchical key-value storage. We use PathStore as the baseline and tailor it to meet the mentioned demands of a streaming file system. The modifications involve changing PathStore's update propagation policy to achieve the desired latency for real-time streaming. Our work exhibits improved latency, as well as PathFS' performance comparison with various baselines

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

edge computing, storage systems, data propagation, hierarchical topology

1 INTRODUCTION

WWith the ever-growing role of computers in daily life, resource and latency demands have taken new shapes. The integration of IoT devices, where hundreds of devices should access real-time data, has brought about a new paradigm in infrastructural design. That is, the traditional cloud computing platforms will hardly meet these demands as they consume excessive bandwidth and limit scalability. Therefore, edge computing has emerged as a new platform to address these challenges. Edge computing suggests breaking down computation tasks between heterogeneous nodes in the network. For example, tasks that are lightweight but are latency-critical can be assigned to nodes closer to the user, namely edge nodes. These edge nodes might have limited resource capacities, yet they are advantageous as they consume little bandwidth and can provide results at lower latency. In edge computing, network nodes often break down into three categories. First, cloud nodes are powerful computers with abundant resources located far from the end-user, and, hence communication will be costly and time-consuming. Second, core nodes are ones located close to network cells or routers, also called cloudlets. Third, edge nodes are those closest to the user with the most scarce resources.

A motivating scenario is a factory line where cameras and sensors continuously produce data, and human operators at other facilities should monitor that data with low latency. Moreover, aggregated data is sent to powerful cloud servers for analytical purposes, where latency is of low importance. Therefore, to accommodate various demands, a suitable file system should address the heterogeneity of resources. Most existing state-of-the-art file systems such as IPFS [6], MooseFS [4], and HDFS [10] do not address that unique characteristic of edge scenarios and assume homogeneous resources, which makes them unsuitable for such environments.

In this paper, we introduce PathFS, an edge-specific file system that supports streaming files. We built PathFS on top of PathStore [9], a key-value storage system for edge computing. PathStore has a hierarchical structure where cloud nodes are located at the root, and edge nodes are the leaves. Therefore, as we go down the hierarchy, resources will be more limited, while communication costs will decrease.

PathStore relies on lazy propagation of updates, which only happen at specific periods over time. This marks a significant shortcoming to support streaming files. Therefore, we modified the PathStore core system and implemented the event-driven system that undertakes eager propagation of updates. That is, data is propagated once it has been inserted. Moreover, we defined a schema to store and represent file system data and integrated that schema with FUSE so that the file system will be accessible through standard operating system tools such as file explorer or terminal.

The remainder of this paper is structured as follows: First, we briefly review the related work and explore PathStore as the background of our work. Then, we explain our contributions and evaluate them through various experiments. Finally, we discuss the results and possible directions for future work.

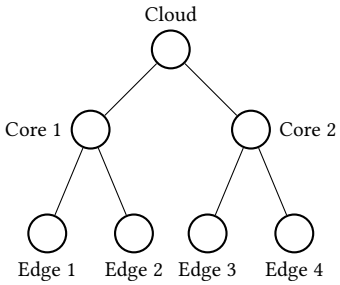

Figure 1: An example of PathStore's hierarchical topology

2 RELATED WORK

Several works have been done on file systems and streaming. For example, IPFS [6] and Ceph [12] are widely used distributed file systems that offer peer-to-peer replication. Moreover, Anna [13] provides a highly scalable key-value store that relies on causal consistency. In terms of streaming file systems, TokuFS [1] is a non-distributed file system tailored for streaming. Other work addresses the problem of HD streaming in distributed environments as well [7]. Despite the various works, they mainly address ordinary distributed systems, not edge environments where different nodes have entirely different capacities. This unique characteristic of edge environments deserves independent attention to create systems that exploit its benefits.

3 BACKGROUND

In this section, we describe the underlying structure of PathStore, on which we implemented PathFS. PathStore has a hierarchical structure in which each node has one parent and multiple children. In a hypothetical topology depicted in figure REF, the root node is a cloud server, while its children are network nodes (also called core nodes or cloudlets). The third level consists of edge nodes that can represent various edge and IoT devices.

Each node is connected to a separate ring of Cassandra to store data. In fact, PathStore is a system that connects and synchronizes multiple Cassandra rings from various data centers that could potentially be far away from each other. In the rest of the paper, we use the term *storage* to refer to each node's corresponding Cassandra ring. PathStore follows the same query language interface as Cassandra, namely CQL. Therefore, records are defined as Cassandra tables, data is stored as Cassandra rows, and is retrieved through CQL queries. However, PathStore slightly modifies the incoming queries to add some specific metadata that is required for its internal functionalities. Examples are parent node ID and insertion time. For simplicity, PathStore assumes that the CQL rows are immutable. This helps PathStore manage the synchronization issues between nodes. Therefore, PathStore turns update and delete queries into insert statements. Then, upon retrieving data, PathStore iterates over the returned rows and builds the resulting row. However, this procedure looks transparent to the end-user, and the end result is the same as the case the updates or deletes were actually executed.

Since PathStore is designed for edge environments, it has the general assumption that nodes on higher levels of the topology are expected to have more resources than the leaves. Therefore, the eventual consistency model ensures each node only stores a subset of its parent's data. Thus, nodes synchronize their data with their parent node *periodically*. That is, at each period, nodes send all data that has been added within the last interval to their parent. On the other hand, they only receive a subset of the corresponding data of the parent node. In other words, they only receive the relevant rows that they are interested in. This concept is implemented in PathStore as *interest sets*. Each node stores the set of select statements clients in its subtree have already executed, and at each period, it re-runs those queries on the parent node's storage. There, according to the insertion time meta column, it only fetches new data that is relevant to itself or its children. Moreover, upon creation, new interests are passed up until reaching the root node so that the consistency conditions are met.

4 IMPLEMENTATION

In this section, we describe PathFS' implementation, which includes (i) the schema to represent and store file system data, (ii) the event-driven update propagation system that is implemented to the core implementation of PathStore, and (iii) the file system interface that is created using the FUSE [11] library so that files would be accessible through regular OS file explorers or terminals.

4.1 The Schema

Since PathStore relies on Cassandra as the sole storage system, data is represented, inserted, and retrieved using Cassandra Query Language (CQL) queries. Therefore, PathFS follows the same rule, and data should be stores CQL rows. For file system purposes, we need to store two separate sets of records for files: The metadata of the entries (which are files, directories, and symbolic links) and the actual content of files. Metadata includes the name of the file, its parent directory, creation time, owner, permissions, etc. Hence, we created two Cassandra tables, namely *Entry* and *FileData*. The full set of attributes of each table are shown in Table 1. In this table, `time` data type is the contracted form of `timestamp`. Moreover, the `follow` attribute is the actual path a symbolic link is referring to. The *Entry* table resembles the inode data structure in Unix file systems, with the exception that it saves the file path (filename and parent) as well. Therefore, our schema does not support hard links.

Furthermore, the *FileData* table is designed to store various chunks of a file. For example, a camera that repeatedly streams captured frames saves the incoming frames as a new chunk of a large file containing all frames. This way, the readers can query for the new chunks (i.e., chunks whose `insertTime` is later than that of the last fetched chunk) and get updates soon with a single query. The `rawData` attribute the actual content of the file stored as byte arrays or blob objects.

4.2 Event-driven Update Propagation

As described in section 3, PathStore propagates updates lazily. That is, nodes periodically sync their data storage with their parents by pushing the new updates to parents and pulling relevant queries from it. This way, eventually, all nodes will have synchronized data. Given PathStore's consistency rule, where each node must store a subset of its parent's data, the push phase involves sending all new

Entry		FileData
parentID: UUID	size: int	fileID: UUID
name: varchar	created: time	insertTime: time
ID: UUID	modified: time	rawData: blob
mode: int	changed: time	
owner: int	blocks: time	
group: int	follow: time	

Table 1: PathFS data representation schema

updates to the parent. However, the pull phase includes fetching the data that the node is subscribed to, which was called the *interest set* in section 3.

The lazy propagation system provides benefits such as moving the propagation tasks to the background, reducing the computation load of the nodes, and batching the incoming updates. However, this system is not suitable for streaming as readers want to access the new chunks as fast as possible. Hence, we devised the event-driven strategy to propagate the updates eagerly. In these systems, updates are propagated as soon as they are successfully written to the source node's storage. The process involves upward and downward propagation that will be described in the following subsections.

4.2.1 Upward Proapagation. Given PathStore's consistently model, each node's parent must eventually store all of the child node's data. Therefore, the upward propagation algorithm is pretty straightforward. That is, the query string, alongside its encoded values, is sent to the parent through a communication channel. Once a node receives such a message from its child, it applies the query to its own storage. Moreover, it uses the same mechanism (upward propagation) to forward the query to its parent. It also invokes downward propagation to forward the incoming query to its children other than the original sender node (i.e., the sender node's siblings in the topology.)

4.2.2 Downward Propagation. The downward propagation algorithm aims to send the query to relevant children. Unlike upward propagation, it should *select* which children should receive the incoming query. That is because if we propagate each piece of data to each child, the consistency model will be full replication, which is not suitable for edge computing as edge nodes might not have the storage and computation capacity to store all data. Instead, we use the *interest* notion that was implemented in PathStore to determine which children should receive the incoming query.

The interest set is the set of select queries a node has subscribed to. This set includes all select queries that have been executed on this node or its children. PathStore already provides interest propagation, so each node's interests will be created upon clients' request and propagated to its ancestors. Therefore, in PathFS, we used this tool to determine if an incoming insert query should be sent to a particular child. Since the child's interest set indicates all select statements that the node or its subtree has executed, the incoming insert query is relevant to that child if it matches at least one of those select statements. Moreover, we only address insert queries as PathStore transforms update or delete statements into inserts, a design choice that was explained in section section 3.

Algorithm 1 Downward Propagation Algorithm

Input: the incoming insert query
1: Execute the insert query to the aux table
2: **for** each subscribed interest **do**
3: Execute the corresponding select query on the aux table
4: **if** the newly added row is returned **then**
5: Forward the insert query to subscriber node by invoking Downward Propagation on that child node.
6: **end if**
7: **end for**
8: Execute the insert query on the actual table

To perform the matching operation, we create an auxiliary table for each of the original tables. Then, we perform the insert operation to that auxiliary table and then execute the select queries of the child's interest set. If the newly added row is returned by at least one of these selects, then it indicates that the insert is relevant to that child and should be forwarded downwards. Afterward, the newly added row will be removed from the auxiliary table to keep it lightweight. For example, let us assume to following queries as the subscribed interests of a particular child.

(1) `SELECT * FROM FileData WHERE insertTime >= 2020-12-29 12:00:00`

(2) `SELECT * FROM FileData WHERE fileID = 35cf5604-9000-46fc-accb-1eeaccf18afa`

Now, let us imagine the following insert query has arrived at the parent.

```
INSERT INTO FileData(fileID, insertTime, rawData) values
(35cf5604-9000-46fc-accb-1eeaccf18afa, 2020-10-10 00:00,
'hello')
```

When downward propagation is invoked, the node executes the insert query into the isolated auxiliary table and then iterates over the two subscribed select statements to see if there is a match. When executing the second query, the newly added row is returned. Therefore, it means that the child should receive updates on the results of that select statement, where the new insert query is a potential result. So it will be sent to the child node and invoke the downward propagation recursively until it reaches every potential subscriber. This procedure is shown in algorithm 1 as well.

4.3 FUSE Integration

Finally, in order to make PathFS accessible through regular operating systems tools (e.g., file explorer, terminal), we implemented PathFS integration with FUSE [11]. FUSE is a kernel module that forwards file-system-related syscalls of the mounted directory to the custom file system (which is PathFS in our case. For integration, one must implement a set of syscalls such as `open, create, readdir, write, read, etc` and mount the file system on a directory. Afterward, the files can be accessed through a regular file explorer or terminal, where the interface is identical compared to

the native file system. In this work, we developed the integration as a two-layer application. The core layer is a regular PathStore client application that stores and retrieves data using the schema defined in section 4.1, and the other one is the FUSE-related syscalls that connect to the core layer and return the results to the FUSE utilities.

5 EVALUATION

In this section, we present the experiment results for evaluating PathFS' performance. These experiments fall into two categories. The first set of experiments measure the performance improvements caused by the event-driven system, while the second set examines the performance of the schema and FUSE integration.

5.1 Event-driven System's Performance

To assess the event-driven system's performance, we deployed PathFS on a five-node emulated topology depicted in fig. 1. We simulated inter-node latencies based on the real-world latency between AWS data centers [2]. Beside PathFS, we deployed the traditional PathStore, as well as IPFS [6] as comparison baselines on these nodes.

Since PathFS' core only differs from PathStore in the development of the event-driven system, other procedures, including defining the tables and connecting the client application, were exactly the same between PathFS and PathStore. However, we simulated the scenario streaming by repeatedly adding a new file to an existing folder and updating the DNS record associated with the folder each time, a process described in [8].

In our experiments, the writer node is located in Montreal, while the reader is in Seoul. Every five seconds, the writer adds a new 16KB or 256KB-sized piece of data. Then, we examine when this data is accessible by the reader node, which marks the overall propagation latency in our topology. The writer node inserts new chunks fifty times.

Experiment results are shown in Figure 3. For 16KB-sized chunks, PathFS' average latency was 511.54ms, while that of PathStore and IPFS were 3471.56 and 1473ms, respectively. Moreover, for 256-sized chunks, PathFS had 1455.2ms latency, while PathStore and IPFS had 4523.9 and 2143.48ms, respectively. Therefore, in these experiments, PathFS showed 80% and 60% improved latency compared to

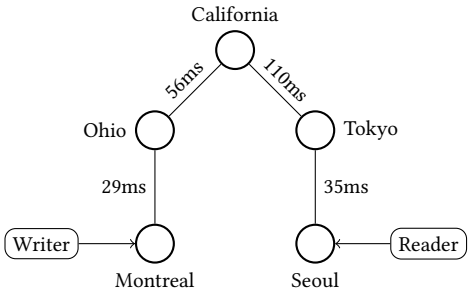

Figure 2: Experimental topology for assessing the event-driven system

PathStore. Also, it was 60% and 33% faster than IPFS as a real-world distributed file system. Moreover, as shown in
Creffigure:latency, PathFS latency was much more steady than PathStore and even IPFS. For PathFS, the high variance in propagation delays is attributed to the nature of push and pull servers that run on certain points in time, so the latency depends on how soon those servers will propagate the data.

5.2 Schema and FUSE Integration Performance

The second set of experiments addressed evaluating the performance of our schema and the FUSE integration. To eliminate other possible factors, we reduced the setup to a single-node topology so that we could make sure that the results are direct causes of the schema or FUSE integration rather than propagation latency or inter-node communication delays.

Even without propagation costs, our system has severe overheads introduced by its various components. These overheads include FUSE overhead, Cassandra overhead, and PathStore overhead which are still significant in the single-node topology. PathStore overheads include an additional process that is done on each query, such as transforming updates and deletes to inserts, as well as its implications of data retrieval.

To assess these factors, we compared PathFS with three baselines. First, to examine PathStore overheads, we deployed the schema and FUSE integration directly on Cassandra. Moreover, we compared PathFS against the native file system and single-host network file system (NFS) to assess its performance against kernel codes. We used the Fio [3] benchmark for all experiments, where the test case involved various random read/writes to a 100MB-sized file. In all experiments, the block size was set to 4KB. That is, writes were writing 4KB of data at once, and reads were requesting a consecutive, 4KB-long part of that file. We measured the average read/write bandwidth or speed, which is a common indicator of file systems' performance.

Experiment results can be found in Figure 4. The nodes where the experiments were carried out used hard disk drives and were not particularly fast. Moreover, the small block size contributed to the relatively low bandwidth obtained in all those experiments. However, comparing PathFS with the Cassandra PathFS baselines, there is an evident 50% overhead introduced by PathStore. Furthermore, comparing the bandwidth results with NFS, the overall performance seems to be promising. However, it is worth noticing that since the host machine used hard drives, it put NFS at a huge disadvantage against PathFS, which is an in-memory file system.

6 DISCUSSIONS

As shown in the experimental results, the event-driven system provided significant improvement in propagation latency. However, eager propagation of all updates is not a feasible choice. That is, the original PathStore's lazy propagation pushed this task to the background and accumulated all updates into batches. Therefore, clients would directly write to each node's storage without passing the request through the PathStore node. However, in the eager system, each query is being sent immediately, limiting the scalability of the system. In other words, the communication channels between

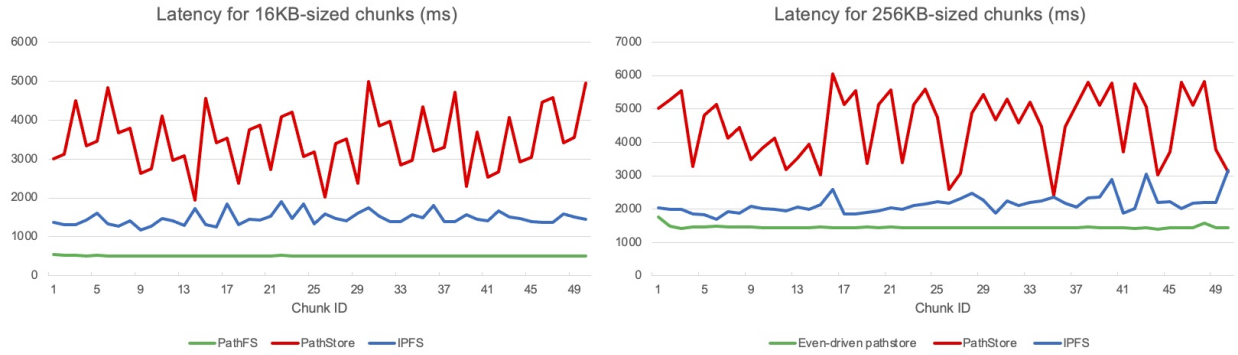

Figure 3: Propagation latency comparison between PathFS, PathStore, and IPFS

FIO Benchmark (random read/ writes)	PathFS	Cassandra pathfs	native	nfs (1 host)
read	629 KB/s	1.2 MB/s	1.8 MB/s	621 KB/s
write	230 KB/s	444 KB/s	666 KB/s	227 KB/s

Figure 4: Experiment results for evaluation the performance of the schema and FUSE integration

nodes have limited bandwidth, and the potential TCP overheads maybe paid off while sending small queries multiple times. Possible solutions could be initiating various communication channels or even turning each PathFS node into a distributed process itself. However, since edge computing often assumes resource scarcity at the edge, these assumptions are not valid. The alternative solution would be prioritizing the incoming updates, where only high-priority updates will be propagated eagerly, and the rest will be handled in the old PathStore's lazy way. The priority assignment strategy and handling the limited inter-node bandwidth marks an appropriate direction for future work.

Moreover, even though the schema results were promising, it was still far from the performance of modern-day file systems. The small block size, as well as the archaic magnetic drives on the test machines, account for the bad performance of all baselines. However, one major limitation of PathFS is PathStore's dependency on Cassandra, which is not known for storing and retrieving large rows. Therefore, as the block size increases too much, the performance is expected to wane, while other baselines are expected to boost. For future work, we can explore the alternative choices for the storage service such as ScyllaDB [5], the C++ implementation of Cassandra, and assess any potential optimization they could offer.

7 CONCLUSIONS & FUTURE WORK

As ubiquitous computing and IoT gain an ever-increasing role in our lives, the traditional cloud systems will hardly meet the demand for fast response times and low bandwidth consumption. Edge computing provides an alternative choice: breaking down computation tasks and migrating less-resource-intensive tasks to edge nodes located close to clients.

This paper presented PathFS and a streaming file system suitable for edge computing built on top of PathStore, a key-value storage system for edge environments. We retained PathStore's hierarchical structure where cloud nodes were generally at the root and edge nodes located at the leaves of the tree. Each node synchronizes itself with its parent, sending all of its updates upwards while receiving only the relevant updates it has subscribed to. To support streaming, we modified PathStore's update propagation system and implemented eager propagation, including upward and downward propagation, where data is sent immediately to the parent and relevant children. Moreover, we designed a schema to store file system data and integrated PathFS with FUSE to make it accessible as a regular OS file system. Our experimental results showed a 60% improvement in terms of propagation latency and also a promising comparison between various existing baselines.

For future work, we intend to address the limitation of the inter-node communication channels, as well as more rigorous testing of the FUSE integration to assess the extent to which each layer is slowing down the system and finding out the performance bottleneck. Moreover, our current system only supports append-only updates as each new data is directly written to the storage as a separate row with the latest `insertTime` value (more details are shown in table 1). Supporting edits and multi-writer scenarios will entail addressing conflict-handling challenges that we intend to explore in the future.

REFERENCES

[1] 2012. The TokuFS Streaming File System. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 12)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/Esmet

[2] 2021. *AWS Latency Monitoring*. https://www.cloudping.co/grid

[3] 2021. Fio. https://fio.readthedocs.io/en/latest/fio_doc.html

[4] 2021. *MooseFS*. https://moosefs.com/

[5] 2021. ScyllaDB. https://www.scylladb.com

[6] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561 http://arxiv.org/abs/1407.3561

[7] Hancong Duan, Wenhan Zhan, Geyong Min, Hui Guo, and Shengmei Luo. 2015. A high-performance distributed file system for large-scale concurrent HD video streams. *Concurrency and Computation: Practice and Experience* 27, 13 (2015), 3510–3522.

[8] Sagar Ganiga. 2018. *Streaming with IPFS*. https://medium.com/revotic/streaming-with-ipfs-2145e6df5a4e

[9] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. 2018. Pathstore, a data storage layer for the edge. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 519–519.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[11] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 59–72.

[12] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*.

[13] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044