Container Native FS Interposer

Julia Hua, Jiawei Xiang, Hilario Gonzalez, Juncheng Cao
Mentors: Vasily Tarasov, Alex Merenstein

December 7, 2024

Project Overview

Container Native FS Interposer

- Container Native Works natively with kubernetes
- FS Interposer Intercepts filesystem operations

A **FUSE**[1] based **CSI**[2] plugin providing various testing utilities for Kubernetes applications, include workload *tracing*, workload *metric* collection, *fault* injection, and *throttling*.

[1] Filesystem in USEerspace

[2] Container Storage Interface

What is FUSE

Filesystem in Userspace

Benefits

- Develop virtual file systems in user space
- Can be implemented with any language
- Available in many linux distributions or even other OS

Drawbacks

- Poor performance

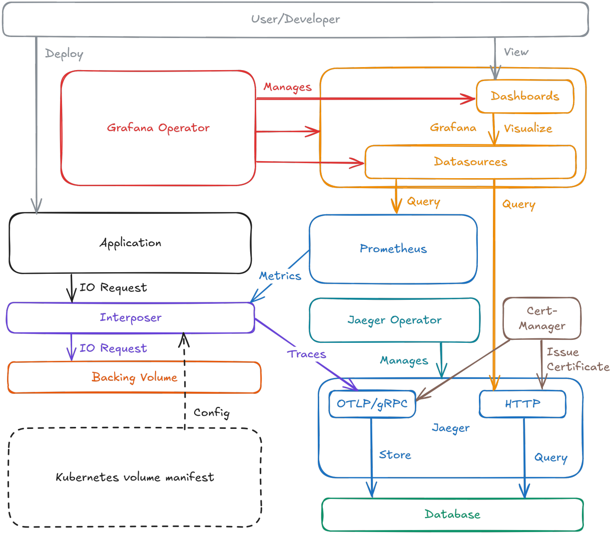
What is CSI

Container Storage Interface

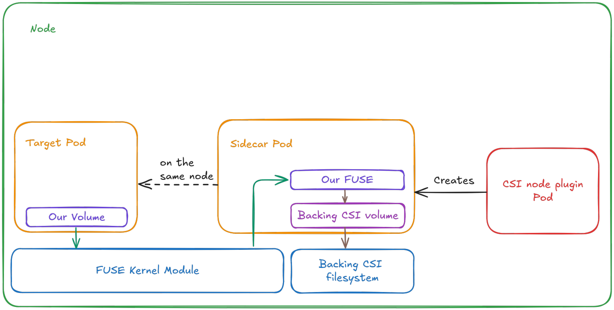
Features

- Container runtime agnostic way to manage storage
- Supported by Kubernetes, Mesos, Nomad, ...
- Provisioning and mounting volumes

Architecture



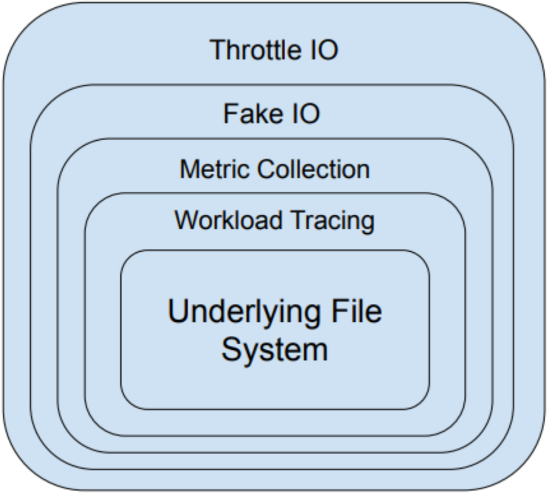
CSI



Sidecar

CSI plugins cannot readily access volumes provided by other CSI plugins, thus requiring the "sidecar" pod as a proxy.

Utility File Systems



Workload Tracing

What

Track the activities of an application under a specific workload

Why

- Find performance bottlenecks
- Debug application
- Analyze system resources used

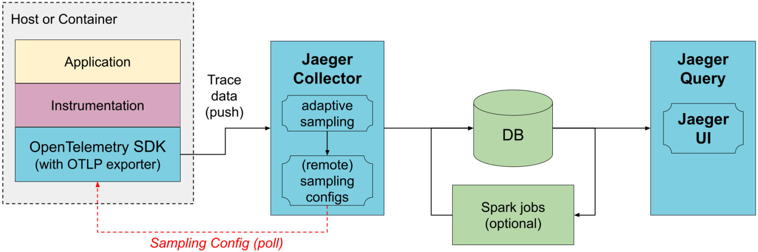
Example Attributes in FUSE FS

- Inode number
- Process ID
- Number of bytes read
- Duration of file operation

OpenTelemetry - Tracing

OpenTelemetry Tracing

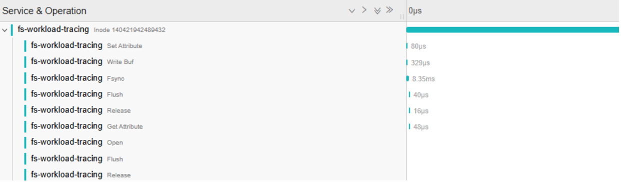
- Trace: Path of your request throughout the application
- Span: Building block of traces, a logical unit of work
- Jaeger backend



Nested File Spans

Nested Spans

- Parent span: File denoted by inode number
- Child span: File operations performed under that file



```
Service & Operation                                   0μs
fs-workload-tracing  Inode 140421942489432
    fs-workload-tracing  Set Attribute                 80μs
    fs-workload-tracing  Write Buf                     329μs
    fs-workload-tracing  Fsync                         8.35ms
    fs-workload-tracing  Flush                         40μs
    fs-workload-tracing  Release                       16μs
    fs-workload-tracing  Get Attribute                 48μs
    fs-workload-tracing  Open
    fs-workload-tracing  Flush
    fs-workload-tracing  Release
```

Metric Collection

What

Gather quantitative and aggregate data on the performance and behavior of an application over time

- Understand type of workload
- Set up alerts when error rate or latency exceeds a threshold
- Monitor performance trends

FUSE File System

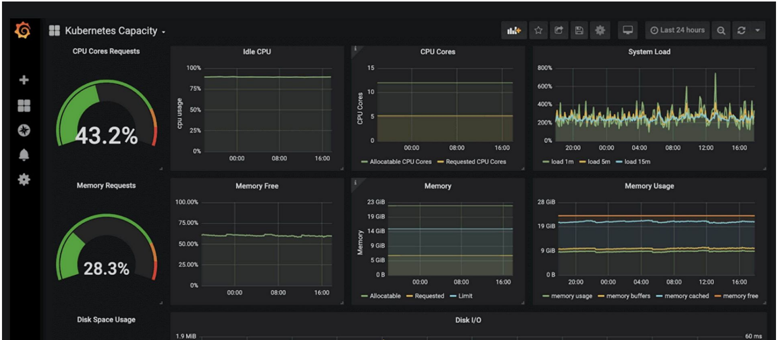
Gather quantitative and aggregate data on the underlying FS.

- The total number of bytes read and written
- Latency distribution
- The net number of directories created

Metric Collection

Metrics vs Tracing

- Metrics: Aggregated, numerical data about a system's performance and workload
- Tracing: Life cycle of individual requests as they move through a system (e.g., a file system)



Fault Injection

What

A FUSE-based system that simulates errors and unexpected behaviors in file system operations. By introducing faults like abrupt exits, delays, or forced errors, it creates a controlled environment to test application resilience. Built on the FUSE low-level API, it enables fine-grained control over file system operations and inodes.

Why

- Validating that applications can withstand faults and maintain data consistency under stress.
- Simulation of faults in cloud native environments helping ensure that applications remain robust against common issues like network latency, storage unavailability, or partial failures in distributed file systems

Faults in FUSE FS

File Faults

- Operations: lo_read(), lo_write_buf(), lo_flush(), lo_open()
- Commands: cat, tail, echo, opening in vim or nano, cp

Directory Faults

- Operations: lo_do_readdir(), lo_opendir()
- Commands: ls, cd, find

Fault Types

- Abrupt exit: EIO, ENOSPC, ENOENT
- Delay
- Truncation

Spans in Fault Injection FS

Structure

- One span generated per low level operation
- Each low level operation adds an event for each fault that was generated in it.

Contents

- Span Attributes
  - Operation name and target
  - Offset (if applicable)
  - Inode Number
- Event attributes
  - Timestamp
  - Error Type
  - Size read/written
  - Delay Time

Configurable Parameters

config.json:

- local_log_path
- file_fail_rate
- directory_fail_rate
- use_seednum
- seed
- delay_time

Throttling

Throttling

Limiting the throughput of IO operations, e.g. the number of bytes written in a given amount of time

Throttling is implemented with the **token bucket** algorithm: every byte read/written *consumes* a given amount of tokens from the bucket corresponding to the file, and the tokens are *replenished* at a set rate.

Final Deliverable

helmfile.yaml

One-click installation of the entire project

Executables

- CSI plugin
- Interposer binary

Interposer binary

- Workload Tracing
- Metric Collection
- Faulty IO
- Throttle IO

Future Work

- Measure and adjust for overhead introduced by FUSE
- Replace FUSE with more performant alternatives (eBPF)
- Display file name instead of inode numbers
- Implement Fake IO utility filesystem

Challenges

Develop environment setup

Use meson wrap to build FUSE together with our project

Managing the deployment of our project and dependencies

Use helmfile and helm to declaratively specify the whole installation

Failures

Opentelemetry SDK installation

Too many components and dependencies

CSI volumes only work on a single node

Distributed applications cannot be instrumented together

IO operations are blocked when throttled

Instead we should return EWOULDBLOCK/EAGAIN

What We Learned

Soft skills

- Design of composable software
- Importance of observability
- Researching and reading documentation
- Agile methodology

Hard skills

- Writing custom filesystem in FUSE
- Instrumenting application with OpenTelemetry
- Basics of kubernetes
- Burndown charts