

Forschungsprojekt Bericht

Gerrit Klein

XXXXXX@hs-esslingen.de

Matr-Nr.: XXXXXX

Studiengang: AIM

SS21 (2021-09-12)

Inhaltsverzeichnis

1. CI-Pipeline	2
1.1. GitHub Actions	2
1.2. Implementierung	3
1.3. Mögliche Erweiterungen	3
2. Dateinamenauflösung	4
2.1. Einarbeitung	4
2.1.1. Tracing	4
Philipp Köster Thesis	4
IOTrace-Analyze Tool	10
2.1.2. InfluxDB: Flux	10
2.1.3. InfluxDB: Tasks	11
Ressourcen für Einarbeitung	12
InfluxDB Entwicklungsumgebung	12
InfluxDB Debugging	13
2.2. Entwickelte Ansätze	14
2.2.1. Web-App	14
Proxy	14
Zusätzlicher Service	15
Client	16
2.2.2. Integration in libiotrace	16
Implementierung	16
Limitierungen	17
2.2.3. InfluxDB	17
Implementierung	17
Umsetzung / Ergebnis	18
Ausblick	21
Verweise	22

1. CI-Pipeline

Um Entwickler ein schnelles Feedback zu geben bzgl. breaking-changes in eigenen commits, wurde eine CI-Pipeline eingerichtet. Diese wurde mittels GitHub Actions direkt in GitHub integriert, wodurch kein extra dedizierter Build-Server erforderlich ist.

In diesem Abschnitt wird die Einrichtung und die grundlegenden Konzepte von GitHub Actions kurz beschrieben. Abschließend werden weitere Optimierungsmöglichkeiten der Pipeline aufgezeigt.

1.1. GitHub Actions

GitHub Actions ermöglichen es alle Tasks des Softwareentwicklungszyklus zu automatisieren [\[gd\]](#). Hierfür muss in dem Git-Repository unter `.github/workflows` entsprechende Workflows im YAML Syntax definiert werden.

Die Workflow-Definition umfasst einen Namen für den Workflow, Event(s), bspw. `push` oder `pull_request`, die diesen triggern und Jobs. Folglich besteht ein Workflow aus einem oder mehreren Job(s). Diese Jobs bestehen wiederum aus Steps, welche schrittweise von Runners abgearbeitet werden. Runners werden bei dem Triggern eines Workflows automatisch für jeden Job bereitgestellt, wobei die Abarbeitung der Jobs standardmäßig parallel erfolgt.

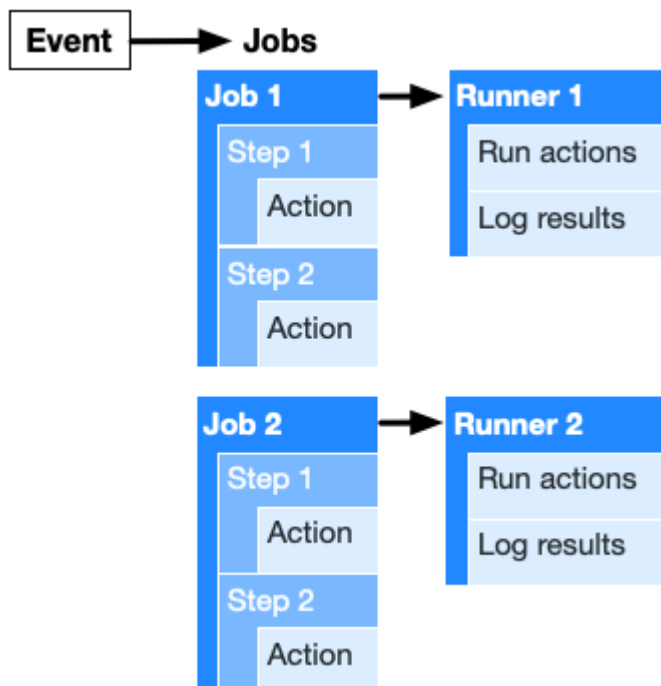


Abbildung 1: Übersicht Komponenten von GitHub Actions (Quelle: [\[gd\]](#))

Die Bereitstellung kann direkt über GitHub erfolgen, es können jedoch auch selbst gehostete Runners verwendet werden. Diese virtuelle Umgebungen werden für jeden Run neu aufgesetzt und können GNU/Linux Distros, Windows oder macOS verwenden. Basierend auf dem GitHub plan gibt es hier jedoch Begrenzungen, so kann bspw. im August 2021 mit Free maximal 20 parallele Jobs ausgeführt werden [\[gb\]](#).

1.2. Implementierung

Die Umsetzung der Pipeline erwies sich initial als sehr umständlich, da das Testen jeder Änderung an dem Workflow ein Commit erfordert, inklusive Wartezeit, bis alle Actions ausgeführt wurden. Um die Umsetzung zu beschleunigen wurde daher lokal mit dem Tool `act` die Pipeline umgesetzt und getestet. Zwecks der lokalen Ausführung muss ein passendes vorgefertigtes Docker-Image verwendet werden, welches alle benötigten Build- und Test-Abhängigkeiten beinhaltet.

Die aktuelle Pipeline umfasst die Jobs `libiotrace` und `iotraceanalyze` und wird bei einem Push bzw. Pull Request getriggert. `iotraceanalyze` baut und testet das IOTrace-Analyze Tool. Der Job `libiotrace` umfasst alle Build-Schritte, beschrieben in `README.md`, um `libiotrace` als dynamically linked library zu bauen. Anschließend werden alle CUnit-Tests mit dem Prefix `test_` ausgeführt. Waren alle Tests erfolgreich, wird ein neues GitHub Release erstellt. Bei gescheitertem Build bzw. Test(s) wird automatisch eine E-Mail an den Committer, der diesen Build getriggert hat, gesendet. Der Status vergangener Workflow runs inklusive Logs kann ebenfalls auf der Projektseite unter “Actions” eingesehen werden.

1.3. Mögliche Erweiterungen

Aktuell wird `libiotrace` lediglich mit den Defaults gebaut, jedoch gibt es einige Compiler-Flags, mit welchen Features aktiviert/deaktiviert werden können. Angesichts der Anzahl von Flags führt dies schnell zu einer kombinatorischen Explosion, die nur schwer mit der Pipeline abdeckbar ist. Es wäre daher sinnvoll die Flags nach Häufig-/Wichtigkeit zu bewerten, um dann die wichtigsten abzudecken.

Weitere mögliche Erweiterungen bestehen in den Tests. Aktuell werden ausschließlich die Tests unter `libiotrace/test/cunit` ausgeführt. Es existieren allerdings weitere Tests unter `libiotrace/test`. Diese müssten einzeln mittels der Shell und `LD_PRELOAD` gestartet werden. Die Schwierigkeit liegt hier zu erkennen, ob ein Test erfolgreich war. Außerdem muss `libiotrace` entweder ohne Live-Tracing kompiliert werden oder, falls mit Default-Flags kompiliert, mit einer InfluxDB Instanz betrieben werden.

Ebenso wird das Live-Tracing aktuell noch nicht abgedeckt. Hierfür müsste zusätzlich ein Docker-Container integriert werden, der eine InfluxDB Instanz bereitstellt. Die resultierenden Daten in der InfluxDB sollten anschließend ebenfalls validiert werden.

2. Dateinamenauflösung

2.1. Einarbeitung

2.1.1. Tracing

In diesem Abschnitt wird die grobe Vorgehensweise und relevante Erkenntnisse während der Einarbeitungsphase zusammengefasst und kurz erklärt.

Philipp Köster Thesis

In der Thesis [\[ki\]](#) wurden grundlegende Prinzipien bzgl. des Tracing erläutert. Die für die Dateinamenauflösung relevante Aspekte sollen folgend aufgegriffen und zusammengefasst werden.



Die Thesis selbst befasst sich ausschließlich mit POSIX-IO.

Grundlagen

Programme verwenden für das Öffnen von Dateien den Dateinamen.

```
// (1) Datei öffnen
int fd = open("/etc/hosts", O_RDONLY);
```

Weitere IO-Operationen ausgehend von dem Programm verwenden anschließend eine ID.

```
// (2) IO-Operation
read(fd, buf, sizeof(buf) - 1);
```

ID-Typen

Diese ID kann verschiedene “Formen” haben. Über Syscalls wie `open`, oben dargestellt, wird ein **File-Descriptor**, auch *Fildes* genannt, als `int` zurückgegeben.

Library-Funktionen aus der libc, erkennbar an dem `f`-Prefix, geben einen **Pointer auf eine FILE-Struktur**, auch *file handle* oder *Stream* genannt, zurück.

```
// (1) Datei öffnen
FILE *fp = fopen("/etc/hosts", "r");

// (2) IO-Operation
fread(buf, sizeof(char), (sizeof(buf) / sizeof(char)) - 1, fp);
```

Zusätzlich kann mittels `mmap` eine bereits geöffnete Datei via dem Fildes in den Arbeitsspeicher gemapped werden. Zugriffe auf diesen Speicherbereich erfolgen über die Startadresse [\[\[ki\], S.4\]](#).

```
// (1) Datei öffnen
int fd = open("/etc/hosts", O_RDONLY);

// (2) In-memory mappen
off_t fsize = lseek(fd, 0, SEEK_END);
void* mmf = mmap(NULL, fsize, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

// (3) IO-Operation
printf("%s", (char*)mmf);
```

Erstellung über Fildes/Stream (anstatt Dateiname)

Es ist möglich ein Fildes von einem Stream zu erstellen und vice versa [\[\[ki\], S.4\]](#).

```
// Fildes -> Stream
int fildes = open("/etc/hosts", O_RDONLY);
FILE* stream = fdopen(fildes, "r");

// Stream -> Fildes
FILE *stream = fopen("/etc/hosts", "r");
int fildes = fileno(stream);
```

Duplizierung von Fildes

Fildes können ebenfalls mittels **dup** bzw. **dup2** dupliziert werden [\[\[ki\], S.4\]](#).

```
int fd = open("/etc/hosts", O_RDONLY);
int fd2 = dup(fd);
```

fd und **fd2** zeigen auf den gleichen Open File Table Eintrag und somit auf die gleiche “Instanz” einer geöffneten Datei.

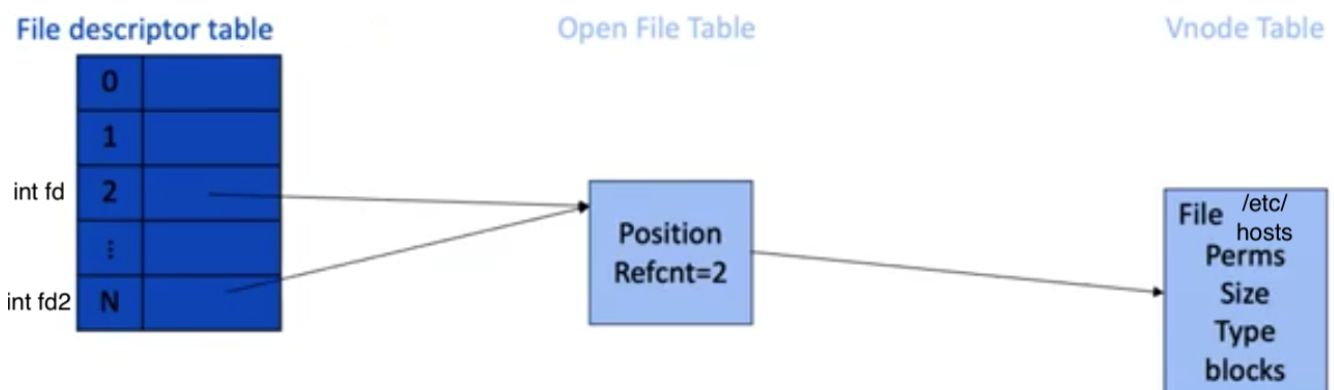


Abbildung 2: Fildes ‘N’ ist Duplikat von ‘2’ (Quelle: Basierend auf [\[fm\]](#))

Forks

Wird ein Prozess geforkt, erhält das Child alle Fildes des Parent mit identischen Integers. Zusätzlich werden die Memory Mappings vererbt. Hierbei wird zwischen privaten (**MAP_PRIVATE**) und public (**MAP_SHARED**) Mappings unterschieden. Bei public wird der gleiche Speicherbereich für beide

Prozesse verwendet, bei `private` eine neue Kopie für den neuen Prozess angelegt [\[\[ki\], S.6\]](#).

```
int fd = open("/etc/hosts", O_RDONLY);
fork();
```

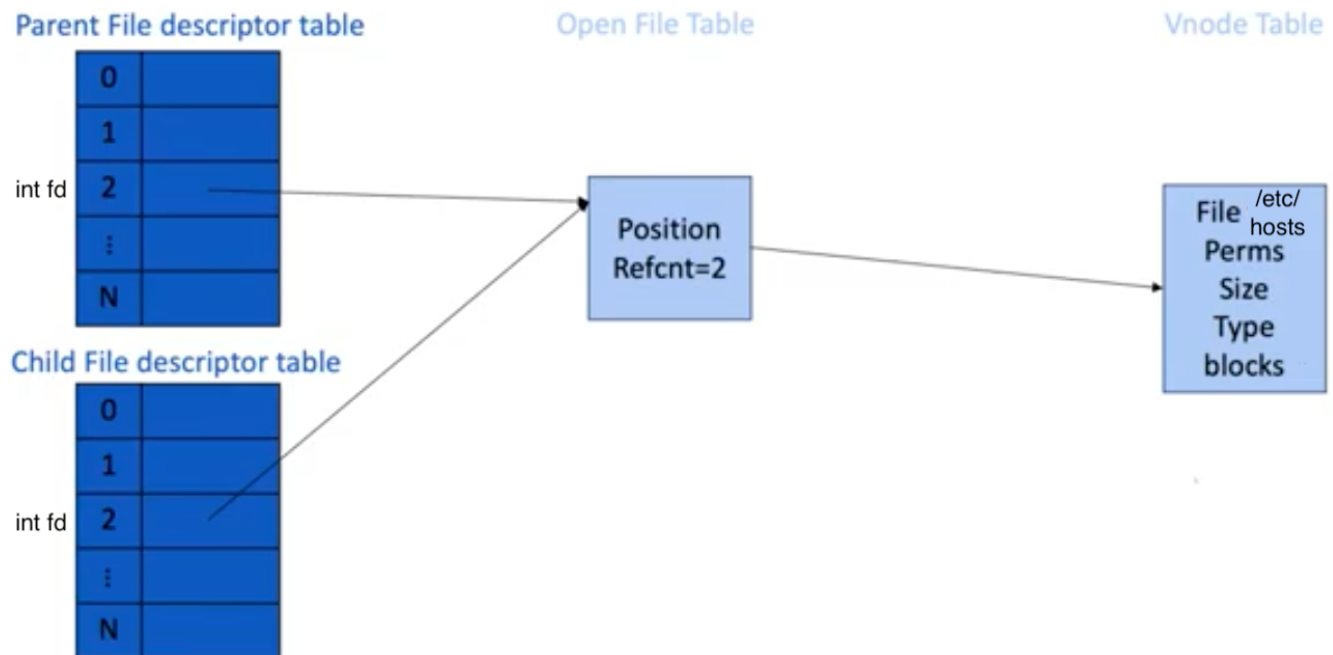


Abbildung 3: Parent und Child haben dieselben Fildes, die auf die gleichen "Instanz" zeigen (Quelle: Basierend auf [\[fm\]](#))

IPC

Prozesse können via Unix Domain Sockets über `sendmsg` bzw. `sendmmsg` und `recvmsg` bzw. `rcvmmmsg` Fildes [\[\[ki\], S.4\]](#) austauschen.

Tracing: Identifikation von Dateien

Für die eindeutige Zuordnung von Dateien in einem Rechencluster wird "das Tripel aus File Serial Number, Device ID und Hostname" [\[\[ki\], S.5\]](#) verwendet.



File Serial Number: Wird auch inode genannt. Eindeutig für jede Datei auf einem Device; kann mittels `stat` oder `fstat` (`sys/stat.h`) ermittelt werden [\[\[ki\], S.5\]](#).



Dateinamen können nicht für die eindeutige Identifizierung verwendet werden. Für jeden Dateinamen in einem Ordner gibt es eine zuordenbare Datei. Jedoch können Dateien mehrere Dateinamen haben. Wie in folgender Grafik dargestellt, können mehrere Hard-Links einen inode referenzieren. Diese können wiederum von beliebig vielen Symbolic-Links referenziert werden. Folglich kann der Dateiname nicht zu der eindeutigen Identifizierung von Dateien verwendet werden [\[\[ki\], S.4-5\]](#).

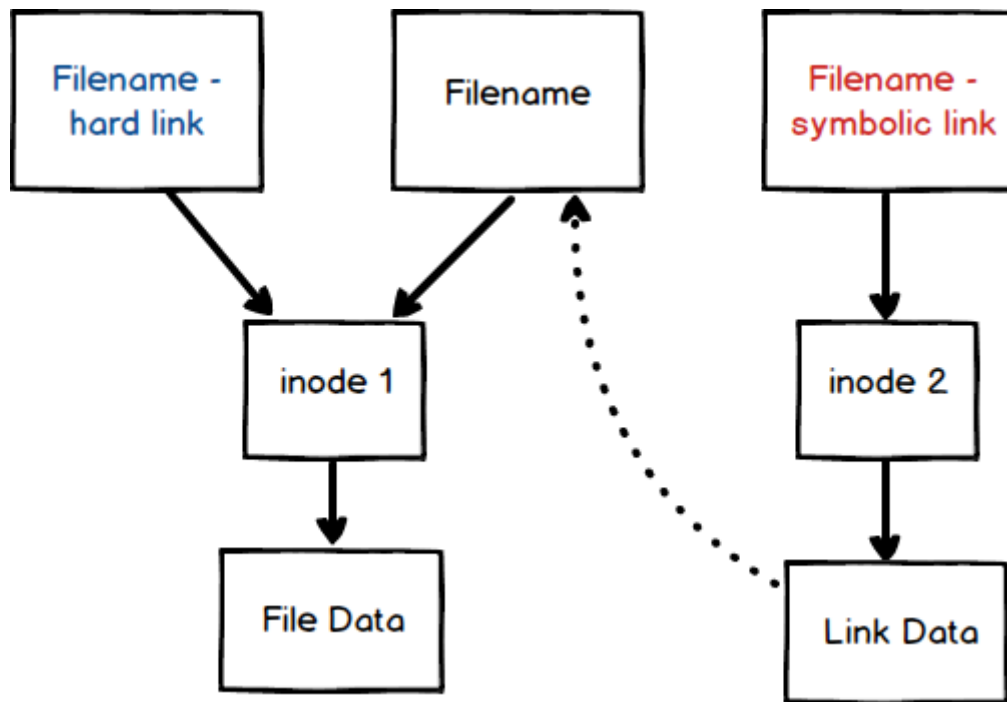


Abbildung 4: Hard- & Soft-Links (Quelle: [\[hs\]](#))

Übersicht Funktionen

In den nächsten Tabellen werden alle relevanten POSIX/OpenMPI Funktionen und die notwendigen Aktionen für das Tracen aufgelistet und beschrieben.

Tabelle 1. POSIX

Funktion(en)	Beschreibung	Aktion
Fildes + Streams		
<code>creat</code> / <code>creat64</code> , <code>mkostemp</code> , <code>mkostemps</code> , <code>mkstemp</code> , <code>mkstemps</code> , <code>open</code> / <code>open64</code> , <code>openat</code>	Öffnen oder erstellen von (temporärer) Datei(en)	Eintrag anlegen; für temporäre Dateien: Als Dateiname generischen Namen generieren (da Dateiname unbekannt ist), z.B. <code>TMP-FILE_<counter></code>
<code>fopen</code> / <code>fopen64</code>	Öffnet Datei und gibt Stream zurück	Gleiches Vorgehen wie bei <code>open</code> (anstatt Fildes wird Stream gespeichert)
<code>tmpfile</code> / <code>tmpfile64</code>	Erstellt temporäre Datei, die automatisch gelöscht wird wenn alle Referenzen geschlossen wurden	Gleiches Vorgehen wie bei <code>mkostemp</code>
<code>freopen</code> / <code>freopen64</code>	Öffnet Datei erneut (falls <code>NULL</code> übergeben wurde) bzw. öffnet andere Datei; die Pointer-Startadresse ändert sich in beiden Fällen nicht	Falls Dateiname (sprich nicht <code>NULL</code>) übergeben wurde → Ersetze ursprünglich zugeordneten Dateiname für Stream

Funktion(en)	Beschreibung	Aktion
<code>close</code> , <code>fclose</code>	Löscht Fildes bzw. schließt Stream	Markiere Fildes bzw. Stream als geschlossen (nur für debugging → eig. nicht erforderlich, da geschlossenes Fildes / Stream nicht nochmals verwendet werden kann (ohne erneutes <code>open</code>))
<code>fcloseall</code> (*)	Schließt alle offenen Streams	Wie bei <code>fclose</code> , jedoch für alle Streams in aktuellem Prozess
“Special” edge cases		
<code>fdopen</code> , <code>fileno</code>	Erstellt für übergebenes Fildes bzw. Stream Stream bzw. Fildes	Neuer Stream bzw. Fildes ZUSÄTZLICH zu bereits gespeichertem Fildes bzw. Stream speichern (beide referenzieren gleichen Dateiname)
<code>dup</code> , <code>dup2</code> , <code>dup3</code> (*)	Dupliziert Fildes (neues Fildes “zeigt” auf gleiche geöffnete Datei)	Neuen Eintrag (dupliziertes Fildes + Dateiname) erstellen
<code>fcntl</code>	Performt verschiedene Operationen auf Fildes basierend auf <code>cmd</code> -Flags	Falls Flag <code>F_DUPFD</code> gesetzt wurde → s. <code>dup</code>
<code>fork</code> , <code>vfork</code>	Forkt Prozess, kopiert dabei alle Streams + Fildes UND public Memory-Mappings (konfiguriert über <code>madvise</code>)	Übernehme alle Fildes + Streams + öffentliche Memory-Mapping für neuen Prozess
<code>sendmsg</code> , <code>recvmsg</code>	IPC: Versenden von Fildes zwischen Parent & Child Prozess via Unix Socket	Array der empfangenen Fildes für neuen Prozess speichern (Achtung: Können umbenannt werden falls IDs bei Empfänger nicht frei waren)
Memory-Mapped-IO		
<code>mmap</code> / <code>mmap64</code>	Mapped Datei in Memory, kann jedoch auch wie <code>malloc</code> für das dynamische allocaten von Memory verwendet werden	Falls bei <code>flags</code> <code>MAP_ANONYMOUS</code> nicht gesetzt wurde → Speichere zurückgegebene Adresse + <code>length</code> unter selbem Dateinamen wie <code>fd</code> ; andernfalls ist das Mapping NICHT file-backed, folglich muss die Zuordnung mit einem generischen Dateiname, z.B. <code>MEM-MAPPING_<counter></code> , gespeichert werden
<code>mremap</code> (*)	Ändert Speicheradresse + Länge eines vorhandenen Mappings	Aktualisiere <code>old_address</code> bzw. <code>old_size</code> zu zurückgegebener neuer Adresse bzw. <code>new_size</code>

Funktion(en)	Beschreibung	Aktion
<code>madvise / posix_madvise</code>	Gibt Kernel Hinweis wie gegebener Memory-mapped Block behandelt werden soll (für bessere Performance/Stabilität)	Relevant für fork: Wurde das Flag <code>MADV_DONTFORK (*)</code> gesetzt → Kopiere Mappings NICHT von Parent zu Child; für <code>MADV_DOFORK (*)</code> (default) → Kopiere Mappings
“Pseudo-Files”		
<code>accept / accept4 (*)</code> , <code>epoll_create (*)</code> / <code>epoll_create1 (*)</code> , <code>eventfd (*)</code> , <code>inotify_init (*)</code> / <code>inotify_init1 (*)</code> , <code>memfd_create (*)</code> , <code>socket</code>	Erstellen “Pseudo-files” (sockets, event notification facilities, anonymous files) → nicht relevant für Tracing	Speichere Fildes + generischen Dateiname, z.B. <code>PSEUDO-FILE_<counter></code> , sodass alle Zugriffe zugeordnet werden können
<code>pipe / pipe2 (*)</code> , <code>socketpair</code>	Erstellen ebenfalls “Pseudo-files”, geben jedoch 2 Fildes mittels Array zurück	Gleiches Vorgehen wie bei <code>accept</code> , jedoch für beide Fildes

Tabelle 2. OpenMPI

Funktion(en)	Beschreibung	Aktion
<code>MPI_File_open</code>	Öffnet eine Datei und gibt ein neues File Handle zurück; Anmerkung: libiotrace generiert für jede geöffnete Datei via <code>MPI_File_c2f</code> ein Identifier, der auch bei folgenden IO-Events (<code>read</code> , <code>write</code>) übergeben wird	Gleiches Vorgehen wie bei <code>open</code>
<code>MPI_File_close</code>	Schließt mit dem übergebenen File Handle assoziierte Datei	Gleiches Vorgehen wie bei <code>close</code>

Immediate- & “Überprüfungs”-Funktionen

Immediate functions (z.B. <code>MPI_File_iread</code> , <code>MPI_File_iwrite</code> , <code>MPI_File_iwrite_all</code> , ...)	Nehmen ein Pointer bzw. Array zu <code>MPI_Request(s)</code> entgegen. Diese struct stellt ein Handle auf Non-blocking Operations dar.	Speichere für jeden Aufruf einer immediate Function die Zuordnung File Handle + Adresse von <code>MPI_Request</code>
“Überprüfungs”-Funktionen	Verwenden <code>MPI_Request</code> handle, um bspw. die Fertigstellung von immediate Functions zu prüfen (mittels <code>MPI_Test</code> , <code>MPI_Testall</code> , <code>MPI_Testany</code> , <code>MPI_Testsome</code> , ...) oder um blockierend auf deren Fertigstellung zu warten (mittels <code>MPI_Wait</code> , <code>MPI_Waitall</code> , <code>MPI_Waitany</code> , <code>MPI_Waitsome</code> , ...).	... Wird später eine Funktion wie <code>MPI_Wait</code> aufgerufen, kann mittels gespeicherter Zuordnung die Dateinamenauflösung erfolgen.



(*) = Linux spezifisch

OpenMPI Funktionsnamen: **all** = Betrifft alle Prozesse, **i** (immediate) = Asynchron
Beispiel: `MPI_File_read` / `MPI_File_read_all`, `MPI_File_iread` / `MPI_File_iread_all`

IOTrace-Analyze Tool

Im Rahmen der Thesis von Hr. Köster wurde ein Analyze-Tool erstellt, welches den Log von einem bereits zur Laufzeit mit libiotrace analysiertem Programm post mortem analysiert.

Hier wurden hauptsächlich Refactorings durchgeführt, um den Code leserlicher zu machen. Zusätzlich wurde das Tool mit in die CI-Pipeline eingebunden.

2.1.2. InfluxDB: Flux

Mit der Einführung der InfluxDB OSS 2 wurde für die Zeitreihen-Datenbank Influx standardmäßig eine neue Query-Sprache “Flux” eingeführt. Diese soll InfluxQL ablösen, welches in InfluxDB 2 nur noch über einen legacy InfluxDB 1 Endpoint unterstützt wird.

Zusätzlich wurde Kapacitor, welcher zusammen mit InfluxDB 1.x als real-time streaming-data processing engine verwendet werden konnte, ersetzt mit sogenannten “InfluxDB Tasks” [ia]. Diese Tasks werden ebenfalls in Flux geschrieben und lösen TICKscript ab, welches von Kapacitor verwendet wurde.

Eine Übersicht aller InfluxDB 2 OSS Komponenten ist in Abb. 5 dargestellt.

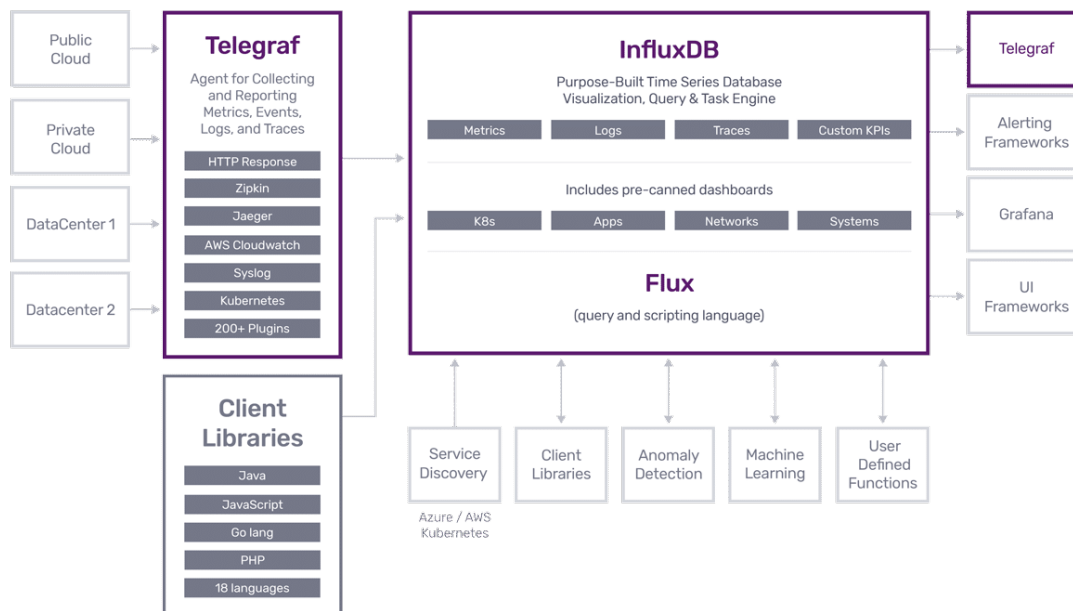


Abbildung 5: Komponenten der InfluxDB 2 Plattform (Quelle: [io])

Flux ist eine funktionale Sprache für das Abfragen von Daten und für das Agieren auf diesen Daten, bestehend aus einer VM und einer Query Engine [it1].

Die Implementierung von InfluxDB 2 folgt laut den Entwickler verschiedene Ziele. So entkoppelt InfluxDB 2 bspw. storage von compute. Flux wird unabhängig von InfluxDB entwickelt und ist daher separat erhältlich [it2]. Dadurch ist Flux, neben InfluxDB 2, als Go library, CLI und REPL

verfügbar und kann als library auch von Dritten in Apps integriert werden. Außerdem ist die Sprache Flux von der Execution Engine entkoppelt, wodurch verschiedene Parser für InfluxQL, TICKscript, Flux, ... geschrieben werden können [\[it2\]](#). Zudem ist Flux erweiterbar, d.h. neben den Built-in Funktionen in der [Standard Flux library](#) gibt es noch weitere Packages, auch von Dritten, die für eine erweiterte Funktionalität importiert werden können. So kann bspw. mit dem [http](#) Package HTTP Anfragen in Flux durchgeführt werden.

2.1.3. InfluxDB: Tasks

InfluxDB Tasks erlauben das geplante Ausführen von Flux Skripts. Diese können aus einer oder mehreren Flux Queries bestehen, welche einen Workflow modellieren.

Ein beispielhafter Task, basierend auf [\[it3\]](#), könnte wie folgt aussehen.

```
// - Task properties -
option task = {
  name: "email alert digest"
  cron: "0 5 * * 0"
}

// - Packages & imports -
import "smtp"

body = ""

// - Flux query: Fetches & aggregates (i.e., transforming data) data pertaining alerts
+ generating mail msg content -
from(bucket: "alerts")
  |> range(start -24h)
  |> filter(fn: (r) => (r.level == "warn" or r.level == "critical") and r._field ==
"message")
  |> group(columns: ["alert"])
  |> count()
  |> group()
  |> map(fn: (r) => body = body + "Alert {r.alert} triggered {r._value} times\n")

// - Send data (Note: Flux supports multiple data sinks) -
smtp.to(
  config: loadSecret(name: "smtp_digest"),
  to: "alerts@influxdata.com",
  title: "Alert digest for {now()}",
  body: message)
```

Wichtig sind hier die Task-Optionen, die beschreiben, in welchem Intervall der Task ausgeführt werden soll. Der Rest des Tasks ist reguläres Flux.

Ressourcen für Einarbeitung

Die Einarbeitung in Influx kann in zwei Teile aufgespalten werden.

Der erste Teil besteht aus der Einarbeitung in InfluxDB und umfasst das Verstehen der “Key concepts” und des “Data Schema” von InfluxDB. Durch die traditionelle Dominanz von relationalen Datenbank ist hier umdenken erforderlich, da viele Konzepte aus Relationalen- nicht auf Zeitserien-Datenbanken übertragbar sind. Ein guter Einstiegspunkt ist hier die [offizielle InfluxData Referenz](#). Zudem stehen auf YouTube einige offizielle Talks von InfluxData zur Verfügung, z.B. [\[it5\]](#).

Generell ist hier Vorsicht geboten, da sich mit dem Release von InfluxDB 2 einiges geändert hat im Vergleich zu Version 1.x. Dies betrifft die verwendete API(s) für das Interagieren mit InfluxDB, die verwendete Query-Sprache, aber auch das Data Schema. Daher beziehen sich viele Diskussionen in Foren oder Talks auf YouTube vor 2018 auf 1.x und sind daher für 2.x teilweise obsolet.

Der zweite Teil der Einarbeitung beinhaltet die Einarbeitung in Flux und InfluxDB Tasks. Ein grundlegendes Verständnis über das InfluxDB Data Schema, z.B. was macht ein Group Key aus, was sind Tags, etc., ist hier vorteilhaft. Als Einstiegsmaterial für Flux ist der offizielle [Getting Started Guide](#) von InfluxData und analog dazu der [Getting Started Guide](#) für InfluxDB Tasks zu empfehlen. Zusätzlich gibt es auf YouTube vereinzelt Talks für Flux Einsteiger, z.B. [\[it6\]](#).

InfluxDB Entwicklungsumgebung

Influx Web-UI

Flux-Skripte können direkt über die Influx Web-UI ausgeführt werden. Syntax Highlighting und Linting werden standardmäßig unterstützt.

Bei zu großen Flux-Skripten kommt es jedoch häufiger zu unresponsive JavaScript, welches unter Firefox den kompletten Tab crashen kann. Folglich sollten in der Web-UI nur kleinere Skripte bzw. Skript Fragmente debuggt werden.

Editor und CLI

Um den Verlust größerer Skripte vorzubeugen empfiehlt es sich lokal mit Visual Studio Code zu arbeiten in Kombination mit der Influx CLI, um diese anschließend auszuführen.

Syntax Highlighting und Autocompletion kann mit dem Plugin [Flux](#) (Extension-ID: `influxdata.flux`) ergänzt werden. Allerdings können Daten nicht wie in der Web-UI mit Graphen dargestellt werden. Ebenfalls unterstützt das Plugin laut Dokumentation das Ausführen und Debuggen von Flux Skripten. Dies scheint jedoch in der aktuellen Version nicht zu funktionieren, da bei der Ausführung von Queries immer folgender Fehler ausgegeben wird: `TypeError: Cannot read property 'split' of undefined`.

Als möglicher Workaround kann für das Ausführen von Skripten die [influxdb-cli](#) verwendet werden. Nach der Installation können die Parameter der Influx-DB Instanz, wie folgt, lokal in der Konfigurationsdatei `~/.influxdbv2/configs` hinterlegt werden, sodass bei späteren Queries mit der CLI nur noch der `config-name` angegeben werden muss.

```
influx config create --config-name local_influx-fspj \
  --host-url http://localhost:8086 \
  --org hse \
  --token OXBW1lU1poZotgyB1Llo2XQ_u4AYGYKQmdxvJJJeotKRyv5m5mjEhCXy0jyldpMmNt_9YY4k3CK-
f5Eh1bN0Ng==
```

Anschließend kann das Skript direkt mit folgendem Befehl ausgeführt werden:

```
influx query --active-config local_influx-fspj --file /path/to/example-query.flux
```

Hierfür kann das in VS Code bereits integrierte Terminal, unter **View** → **Terminal**, verwendet werden, wie in dem unteren Screenshot dargestellt.

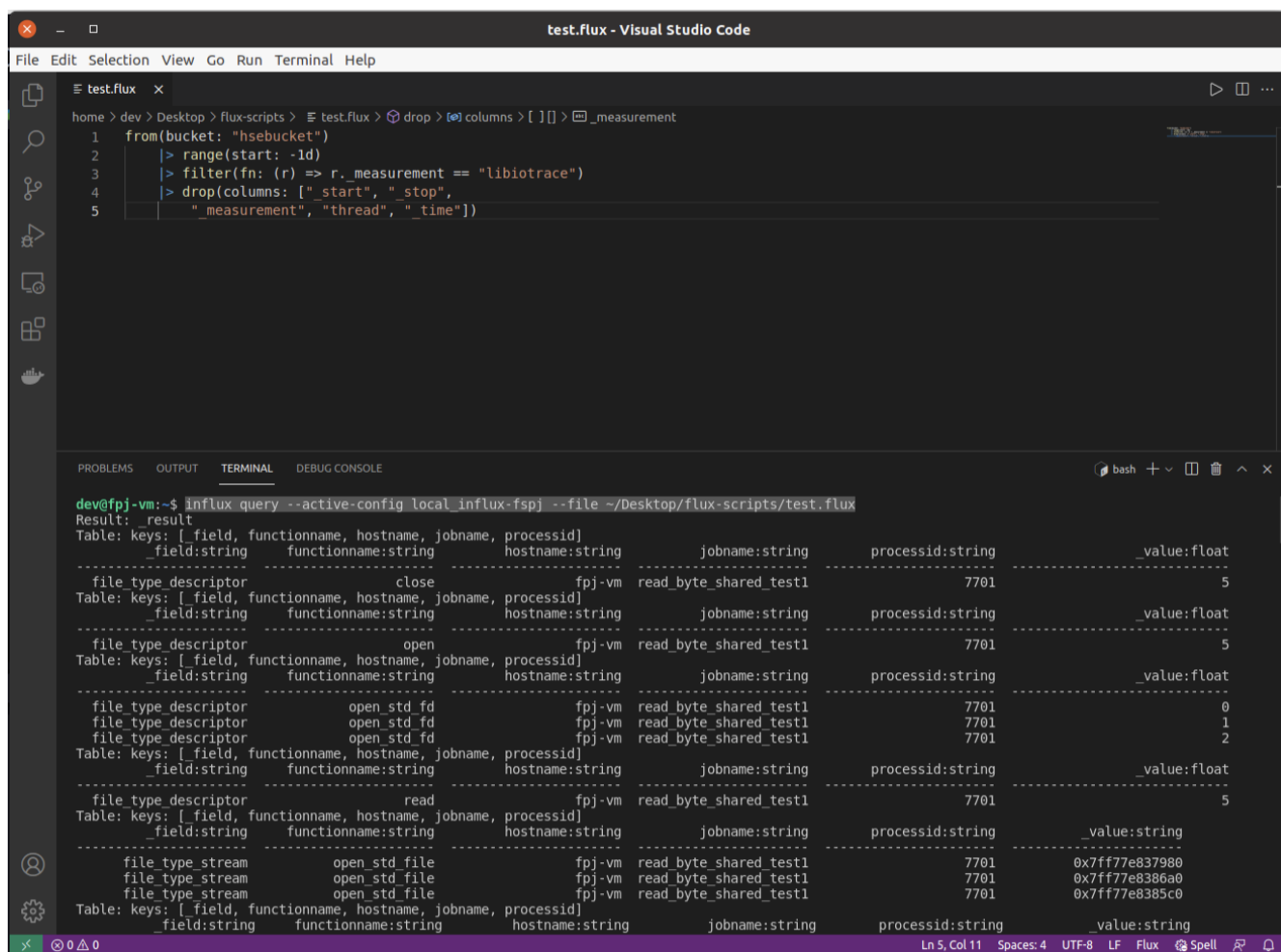


Abbildung 6: Eingerichtete Entwicklungsumgebung für Flux (Quelle: Eigener Screenshot)

InfluxDB Debugging

Aktuell sind die Möglichkeiten bzgl. Debugging von Flux-Skripten sehr beschränkt. Die einzige Möglichkeit aktuell ist das Verwenden von **yield**'s, dem **printf** in Flux, um Zwischenergebnisse von längeren Queries auszugeben.

2.2. Entwickelte Ansätze

In diesem Abschnitt werden die Ansätze für die Umsetzung der Dateinamenauflösung vorgestellt, sowie deren Vor- und Nachteile.

2.2.1. Web-App

Dieser Ansatz beinhaltet die Erstellung einer Web-Applikation, die für das Auflösen der Dateinamen verantwortlich ist. Die Web-Applikation kann anschließend in das `docker-compose.yml` eingebunden werden und somit direkt mit der InfluxDB Instanz gestartet werden.

Unterschiedliche Teilansätze und verschiedene Technologien können hier verwendet werden für die Umsetzung dieses Ansatzes. Ein großer Nachteil, der alle Teilansätze betrifft, ist die potenzielle Out-of-order delivery von Datenpaketen, da jeder Thread einen eigenen Socket für die Kommunikation mit der InfluxDB verwendet. Öffnet bspw. Thread-1 eine Datei mit `open`, welche anschließend von Thread-2 für ein `read` verwendet wird, könnte das `open` nach dem `read` ankommen.

Proxy

Der erste Teilansatz ist die Implementierung als Proxy.

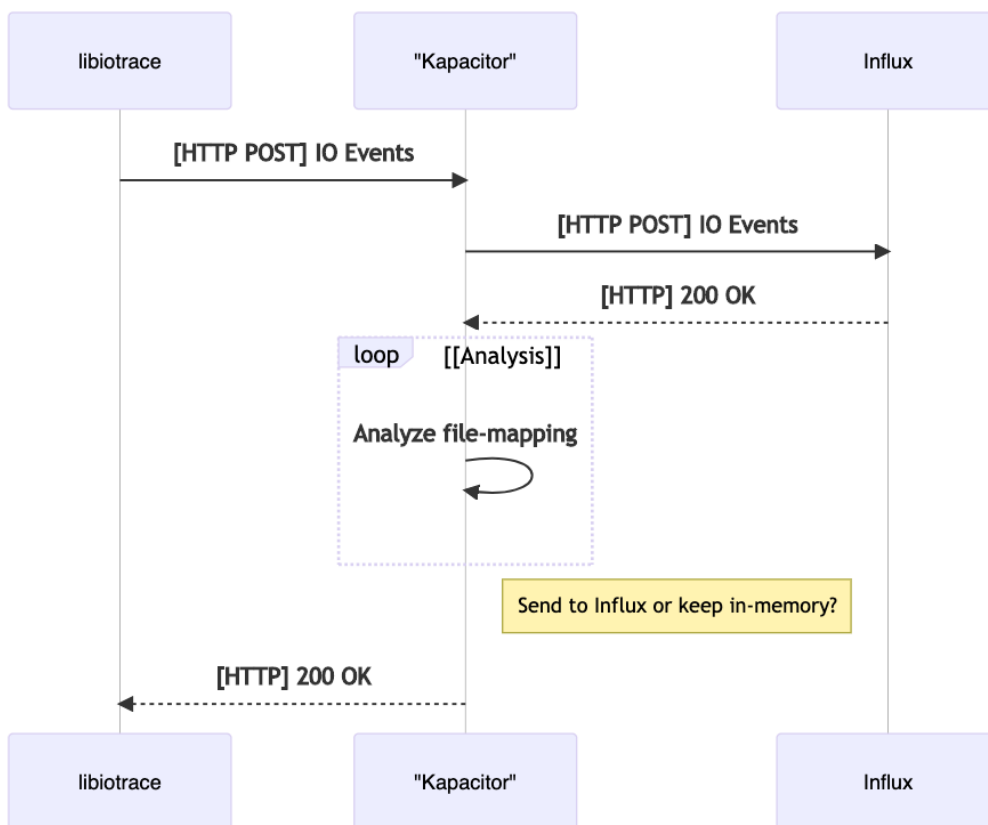


Abbildung 7: Visualisierung des Proxy-Ansatz (Quelle: Eigene Darstellung)

Die Web-Applikation, hier "Kapacitor" benannt, nimmt alle IO-Events entgegen, leitet diese an die InfluxDB weiter und analysiert anschließend alle IO-Events. Diese können anschließend entweder direkt über die Web-Applikation mittels einer API vorgehalten oder direkt in der InfluxDB gespeichert werden.

Dieser Teilansatz ermöglicht Preprocessing, was InfluxDB 2.0 aktuell noch nicht unterstützt [ig1]. Da die Events somit nicht ständig von der DB geladen werden müssen, müssen diese nicht als “abgearbeitet” markiert werden. Ebenfalls fällt der Overhead für das Auflösen von Dateinamen für die InfluxDB weg. Der Teilansatz ist zudem portabel, unabhängig von der verwendeten DB-Lösung. Außerdem muss die libiotrace den Proxy nicht kennen, da dieser transparent agiert, womit keine Änderungen an der libiotrace nötig sind.

Der primäre Nachteil hingegen ist ein zentrales Performance-Bottleneck, welches die Nutzbarkeit des Live-Tracing signifikant einschränken könnte. Zudem ist ein weiterer Komponente, der nicht direkt in InfluxDB oder in der libiotrace integriert ist, notwendig.

Zusätzlicher Service

Um ein zentrales Bottleneck zu vermeiden, kann die libiotrace angepasst werden, sodass nur relevante Events an die Web-Applikation gesendet werden.

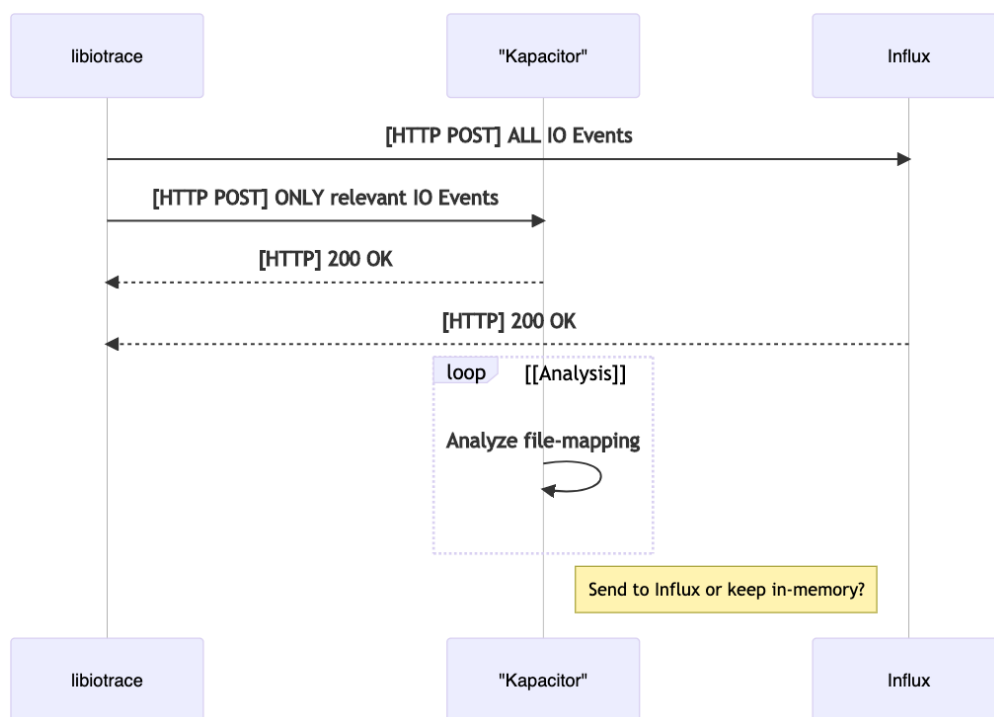


Abbildung 8: Visualisierung des “Zusätzlicher Service”-Ansatz (Quelle: Eigene Darstellung)

Alle IO-Events werden weiterhin an die InfluxDB direkt gesendet. Zusätzlich sendet die libiotrace nun relevante IO-Events an die Web-App. Somit muss die libiotrace einen weiteren Service kennen, daher die Bezeichnung “Zusätzlicher Service” für diesen Teilansatz. Der Overhead des Network-Stack steigt entsprechend ebenfalls auf den Nodes des Clusters, die mit libiotrace überwacht werden.

Client

Ein weiterer Teilansatz, der ebenfalls ein zentrales Bottleneck meidet, ist der Client Teilansatz.

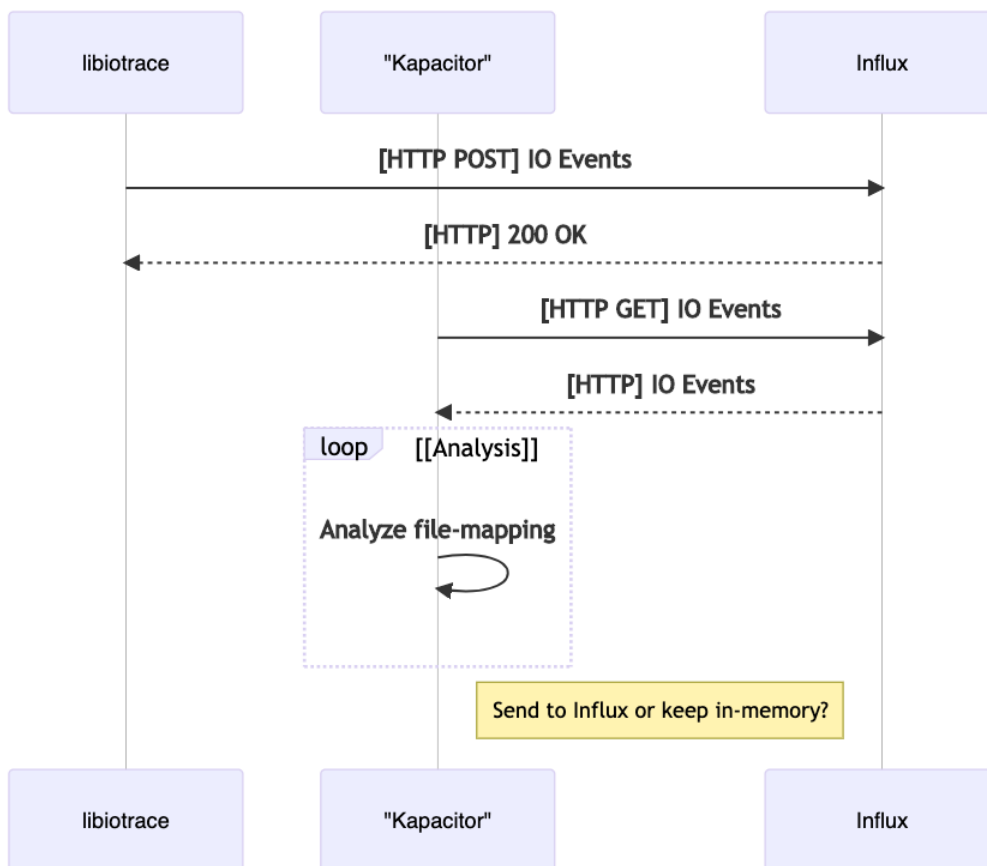


Abbildung 9: Visualisierung des Client-Ansatz (Quelle: Eigene Darstellung)

Die libiotrace kommuniziert nach wie vor direkt mit der InfluxDB, muss somit die Web-Applikation nicht kennen. Die Web-Applikation lädt in festen Intervallen kontinuierlich neue IO-Events von der InfluxDB, analysiert diese und schreibt die Ergebnisse zurück in die InfluxDB bzw. bietet diese via API an.

Notwendige InfluxDB 2 Client Libraries für die Kommunikation zwischen InfluxDB und Web-Applikation stehen [für alle gängige Programmiersprachen bereit](#).

2.2.2. Integration in libiotrace

In diesem Ansatz findet die Erfassung von IO-Events und Analyse zu der Dateinamenauflösung direkt in der libiotrace statt.

Dies hat den Vorteil, dass alle Events in Echtzeit und in chronologischer Reihenfolge abgearbeitet werden können.

Implementierung

Für die Umsetzung muss ein weiteres Modul in die libiotrace integriert werden, welches weitere Komplexität für die libiotrace und Overhead für das zu überwachende System darstellt. Daher soll diese Funktionalität mittels Compiler-Flag deaktivierbar sein.

Für eine einfache Integration soll das Modul aus zentraler Stelle in der libiotrace für jeden Wrapper

aufgerufen werden. Somit müssen einzelne Wrapper nicht angepasst werden. Dies kann bspw. in `wrapper_defines.h` in dem Macro `WRAP_END` erfolgen. Anschließend wird basierend auf den übergebenen Daten, der `basic` struct in `libiotrace_structs.h`, das IO-Event zugeordnet und wie in dem Abschnitt [Übersicht Funktionen](#) beschrieben verarbeitet.

Die Zuordnung der File-Handles und Dateinamen werden darauf in einer Map für die gesamte Laufzeit des zu beobachtenden Programms zwischengespeichert. Somit hat jeder Prozess des zu beobachtenden Programms eine eigene "globale" Map, die alle Threads dieses Prozesses umfasst, womit die Map-Implementierung Thread-Safe sein muss. Hierbei ist es wichtig, Thread-Safety mittels atomaren Instruktionen wie Compare-And-Swap zu garantieren, da Locks zu Deadlocks führen können. Dies ist der Fall, wenn Threads sterben, bevor alle gehaltenen Locks zurückgegeben werden konnten.

Der Key für die Zuordnung sollte das File-Handle selbst, den Typ und die Länge beinhalten. Das File-Handle ist die Speicheradresse bzw. der Integer und kann vom Typ Stream, Memory-mapping bzw. Fildes sein. Zusätzlich zu der Speicheradresse bzw. Integer muss der Typ des File-Handle gespeichert werden, um das File-Handle später mit dem richtigen Typ verwenden zu können. Für Memory-mappings soll zudem die Länge des Mappings gespeichert werden.

Das Modul ist ebenso zuständig für das Auflösen und Ergänzen der Dateinamen für jedes IO-Event. Dazu wird das Handle in der Map nachgeschlagen. Wird ein passender Dateiname gefunden, so wird dieser zu den übergebenen Daten ergänzt, welche abschließend von der libiotrace in die Log-Datei geschrieben bzw. an die InfluxDB gesendet werden.

Limitierungen

Prozesse können mittels IPC gegenseitig Fildes austauschen, wie in dem Abschnitt [Übersicht Funktionen](#) beschrieben. Da jeder Prozess eine eigene Map verwendet, ist aus Sicht des versendeten Prozesses lediglich der `sendmsg`- und aus Sicht des Empfängers lediglich der `recvmsg` Aufruf sichtbar. Somit kann in dem Empfänger-Prozess keine Zuordnung für das empfangene Fildes gespeichert werden, da der Dateiname selbst nicht mitgesendet wird und ein Zugriff auf die Map des Versender-Prozesses nicht möglich ist.

2.2.3. InfluxDB

In diesem Ansatz wird die Auflösung von Dateinamen direkt in die InfluxDB integriert.

Dies hat den Vorteil, dass keine Änderungen an der libiotrace erforderlich sind und Bottlenecks in Form von weiteren Komponenten zwischen der libiotrace und der InfluxDB wegfallen. Ein großer Nachteil, wie in [Web-App](#) beschrieben, ist die potenzielle Out-of-order delivery von Datenpaketen. Eine weitere Beschränkung ist die (noch) fehlende Unterstützung von Stream Processing [\[ig1\]](#), womit InfluxDB 2 aktuell lediglich Batch Processing unterstützt.

Implementierung

Das Ziel ist die Erstellung eines Measurements, welches zeitlich chronologisch alle geöffneten Dateien mit Handle und zugehörigem Dateiname enthält. Vereinfacht dargestellt könnte ein solches Zuordnungs-Measurement wie folgt aussehen.

_measurement	_field	_value	id	type	mmap_len
libiotrace-fns	filename	/etc/passwd	565	F_FIDLES	-1
libiotrace-fns	filename	/etc/resolv.conf	0x561f06b684e0	F_STREAM	-1
libiotrace-fns	filename	TMP_FILE-1	0x424f05c680e0	F_STREAM	-1

Anmerkung: Zwecks Vereinfachung wurden notwendige Tags wie `_time`, `jobname`, `hostname` und `processid`, die für die Zuordnung ebenfalls relevant wären, weggelassen.

Dieses Measurement muss in regelmäßigen Abständen aktualisiert werden mit den neuesten IO-Events, die für File-Tracing relevant sind. Eine Aufzählung relevanter Funktionen ist in [Übersicht Funktionen](#) gegeben. Anschließend muss für alle Events, die auf Dateien bezogen sind, der `traced_filename` ergänzt werden, welcher durch das Nachschauen in dem Measurement ermittelt wird. Abschließend müssen alle bearbeiteten Events als “abgearbeitet” markiert werden.

Grafisch würde der Workflow dann wie in [Abb. 10](#) dargestellt aussehen.

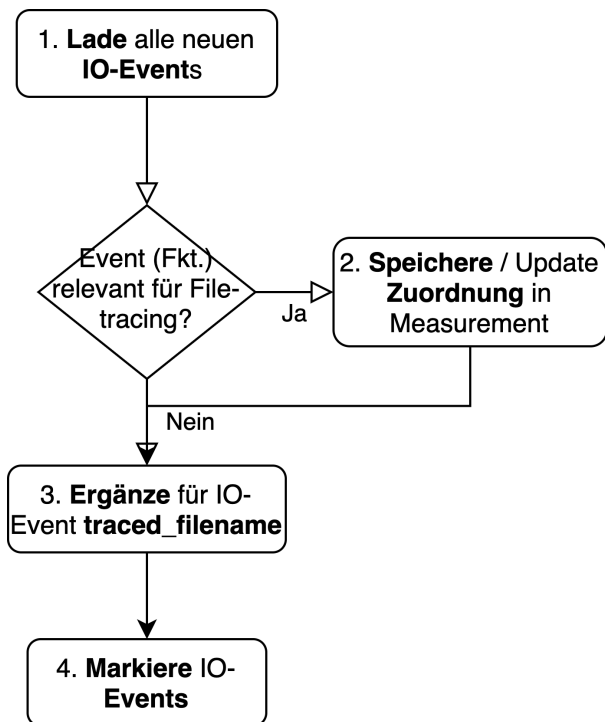


Abbildung 10: Influx Task Workflow (Quelle: Eigene Darstellung)

Werden dann Abfragen in Flux, bspw. für ein Grafana Dashboard, geschrieben, kann einfach auf das Attribut `traced_filename` zugegriffen werden.

Umsetzung / Ergebnis

Die Umsetzung dieses Ansatzes erwies sich kompliziert, weshalb der Ansatz nicht weiter verfolgt wurde.

Die erste Schwierigkeit war gegeben durch die fehlende Unterstützung von Stream Processing. Dadurch müssen Continuous Queries, in Influx 2.x als Influx Tasks bekannt, verwendet werden. Dies hat den inhärenten Nachteil, dass bereits verarbeitete IO-Events mehrfach geladen werden, weshalb ein Filter notwendig ist. Option A wäre filtern durch “markieren” von bereits verarbeiteten

IO-Events mit einem nachträglich ergänztem Tag. Das nachträgliche Ergänzen von Tags scheint jedoch von InfluxDB nicht unterstützt zu werden [ig3]. Option B wäre filtern mit `time-range`, welcher das Query result-set bspw. auf alle Events mit einer `_time` innerhalb der letzten 2 Sekunden beschränkt, bei einer ein-sekündlichen periodischen Task Ausführung. Wird bei dem Versenden der IO-Events an InfluxDB in dem Line protocol der Timestamp ausgelassen, wird dieser von InfluxDB selbst gesetzt [il]. Somit müssten alle IO-Events, die in dem Intervall seit der letzten Ausführung angekommen sind berücksichtigt werden. Um dennoch die chronologische Reihenfolge der IO-Events sicherzustellen, sollten diese anhand der column `wrapper_time_end` sortiert werden.

Eine Implementierung in Flux könnte wie folgt aussehen.

```
// - Task options
option task = {
  name: "resolve_filenames",
  every: -1s,
  offset: 0m,
  concurrency: 1
}

// - Globals
ORGANIZATION = "hse"
DATA_BUCKET = "hsebucket"
MEASUREMENT_NAME = "libiotrace"
MEASUREMENT_MAPPING_NAME = "libiotrace-fns"

// 0. Fetch new events
newEvents = from(bucket: DATA_BUCKET)
  |> range(start: -2s)
  |> filter(fn: (r) => r._measurement == MEASUREMENT_NAME)

// 1. Merge into single "entry" containing all relevant values & tags + sort
irrelevantFields = ["thread", "thread_id",
  "time_diff", "time_end", "time_start",
  "function_name", "process_id",
  "wrapper_time_start"]

sortedFEvents = newEvents
  |> filter(fn: (r) => r._field != "hostname")
  |> pivot(
    rowKey: ["_measurement", "jobname", "hostname", "processid", "thread",
      "functionname", "_time"],
    columnKey: ["_field"],
    valueColumn: "_value"
  )
  |> drop(columns: irrelevantFields)
  |> sort(columns: ["wrapper_time_end"])
```

Im nächsten Schritt, der in [Abb. 10](#) als Schritt 2 eingezeichnet ist, müssen diese `sortedFEvents` gefiltert und nach Funktionsname abgearbeitet werden. Die relevanten Funktionen für diesen Schritt werden, wie eingangs erwähnt, in dem Abschnitt [Übersicht Funktionen](#) aufgelistet.

Eine vereinfachte Implementierung für die Funktion `open` könnte wie folgt umgesetzt werden.

```
// 2. Transform + store all function events in own dedicated measurement
// -> Process all 'open's
openFEvents = sortedFEvents
  |> filter(fn: (r) => r.functionname == "open")
  |> map(fn: (r) => ({                                // Convert types + set
props
    r with
      mmap_len: "-1",
      type: "F_FILDES",
      _time: time(v: int(v: r.wrapper_time_end)),      // ? Produces wrong date ?
      id: string(v: r.file_type_descriptor),

      _field: "filename",
      _value: r.function_data_file_name,
      _measurement: MEASUREMENT_MAPPING_NAME
    )))
  |> keep(columns: ["jobname", "hostname", "processid", "id", "mmap_len", "_field",
"_value", "_time", "_measurement"])
  |> to(                                                // Save as new measurement
    bucket: DATA_BUCKET,
    org: ORGANIZATION,
    timeColumn: "_time",
    tagColumns: ["jobname", "hostname", "processid", "id", "filename", "mmap_len",
"type"]
  )
```

Somit wurde in dem Measurement `libiotrace-fns` die Zuordnung aktualisiert. Im nächsten Schritt, in [Abb. 10](#) als Schritt 3 eingezeichnet, müssen alle andere IO-Events, wie `read`, `write`, etc. mit dem Dateinamen aus der gespeicherten Zuordnung ergänzt werden.

Für `read` könnte eine vereinfachte Implementierung wie folgt umgesetzt werden.

```
// 3. Add traced filename for IO-Events
readFEvents = sortedFEvents
  |> filter(fn: (r) => r.functionname == "read")

fildesFiles = from(bucket: DATA_BUCKET)
  |> range(start: -5d)
  |> filter(fn: (r) => r._measurement == MEASUREMENT_MAPPING_NAME and r.type ==
"F_FILDES")

// ...
```

Die zwei result-sets `readFEvents` und `fildesFiles` müssen mit einem Inner Join gejoint werden, wodurch ein neuer “entry” entsteht, welcher den getrachten Dateinamen enthält. Der Join erfolgt über die columns `processid`, `jobname`, `hostname` und `id`.

Dies ist aktuell jedoch selbst mit Konversion von allen notwendigen Datentypen nicht möglich, was vermutlich auf einen Bug zurückzuführen ist, s. [\[ie1\]](#).

Das eigentliche Hauptproblem liegt aber in der Tatsache, dass Tags nicht nachträglich ergänzt werden können [\[ig3\]](#). Somit müssten die Ergebnisse theoretisch separat gespeichert werden, was zu teilweise redundanten Daten führen würde.

Aufgrund dieser Probleme sowie weiterer Limitierungen seitens Flux wurde dieser Ansatz nicht weiter verfolgt.

Ausblick

Geplante Erweiterungen in [InfluxDB](#) und [Flux](#) für zukünftige Release sind öffentlich auf GitHub einsehbar. Jedoch werden häufig Feature Requests von Nutzern auf die “lange Bank geschoben”, z.B. Preprocessing [\[ig1\]](#) oder Outer Joins [\[ig2\]](#). Für Outer Joins ist das Ticket bspw. bereits seit fast 3 Jahren offen.

Folglich wäre die Empfehlung in der nahen Zukunft komplexere Vorhaben mit anderen Ansätzen / Technologien zu lösen.

Verweise

- [gd] GitHub, Inc.. Introduction to GitHub Actions. URL: <https://docs.github.com/en/actions/learn-github-actions/introduction-to-github-actions>. 2021. Letzter Zugriff: 03.08.2021
- [gb] GitHub, Inc.. Usage limits, billing, and administration. URL: <https://docs.github.com/en/actions/reference/usage-limits-billing-and-administration>. 2021. Letzter Zugriff: 05.08.2021
- [ki] Philipp Köster. Konzeption und Implementierung eines Werkzeugs zur I/O-Performanceanalyse. HS Esslingen. 2020.
- [fm] Margo Seltzer. File Descriptor Management. URL: <https://www.youtube.com/watch?v=p1QuvdP7t4Y>. 2020. Letzter Zugriff: 27.07.2021
- [hs] Cristian Gomez. What is the difference between a hard link and a symbolic link?. URL: https://medium.com/@1154_75881/what-is-the-difference-between-a-hard-link-and-a-symbolic-link-14db61df7707. 2019. Letzter Zugriff: 27.07.2021
- [ig1] gabor. [Feat Req] Support streaming from InfluxDB. URL: <https://github.com/grafana/grafana/issues/28097>. 2020. Letzter Zugriff: 12.08.2021
- [ig2] jlapacik. Add support for different join types. URL: <https://github.com/influxdata/flux/issues/84>. 2018. Letzter Zugriff: 13.08.2021
- [ie1] russorat. panic: column _measurement:string is not of type int. URL: <https://github.com/influxdata/flux/issues/2323>. 2020. Letzter Zugriff: 13.08.2021
- [ig3] mvadu. [feature request] Insert new tags to existing values, like update. URL: <https://github.com/influxdata/influxdb/issues/3904>. 2015. Letzter Zugriff: 13.08.2021
- [io] InfluxData Inc.. InfluxDB Time Series Platform | InfluxData. URL: <https://www.influxdata.com/products/influxdb/>. 2021. Letzter Zugriff: 05.08.2021
- [il] InfluxData Inc.. Line protocol | InfluxDB OSS 2.0 Documentation. URL: <https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/#timestamp>. 2021. Letzter Zugriff: 05.08.2021
- [ia] InfluxData Inc.. InfluxDB 2.0 Open Source is Generally Available | InfluxData. URL: <https://www.influxdata.com/blog/influxdb-2-0-open-source-is-generally-available/>. 2020. Letzter Zugriff: 12.08.2021
- [it1] Paul Dix (InfluxData). Why InfluxDB is Building Flux, a new scripting and query language. URL: <https://www.youtube.com/watch?v=ZjnPNPXXeSo>. 2019. Letzter Zugriff: 13.08.2021
- [it2] Paul Dix (InfluxData). The Design of IFQL, the New Influx Functional Query Language. URL: <https://www.youtube.com/watch?v=asHISf26zTg>. 2018. Letzter Zugriff: 13.08.2021
- [it3] Paul Dix (InfluxData). Paul Dix [InfluxData] | InfluxDB 2.0 and Flux – The Road Ahead | InfluxDays London 2019. URL: <https://www.youtube.com/watch?v=fchF8BqzH88>. 2019. Letzter Zugriff: 13.08.2021
- [it5] Michael Desa (InfluxData). Michael DeSa [InfluxData] | InfluxDB 101 – Concepts and Architecture | InfluxDays SF 2019. URL: https://www.youtube.com/watch?v=GdkW0TmO_xM. 2019. Letzter Zugriff: 13.08.2021
- [it6] Noah Crowley (InfluxData). A Deeper Dive into Flux. URL: <https://www.youtube.com/watch?v=nQf-oB9EQYw>. 2019. Letzter Zugriff: 13.08.2021