Doctoral Program in Informatics (PDINF)

8 April 2024

Flexible Tracing and Analysis of Applications' I/O Behavior

Tânia Esteves

Under the supervision of
Prof. João Paulo
Prof. Rui Oliveira

INESCTEC
HASLab HIGH-ASSURANCE SOFTWARE LABORATORY
Universidade do Minho

Data-centric and Distributed Systems

- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)

Data-centric and Distributed Systems

- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)

Healthcare

Data-centric and Distributed Systems

- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)

Healthcare

Financial Services

Data-centric and Distributed Systems

- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)

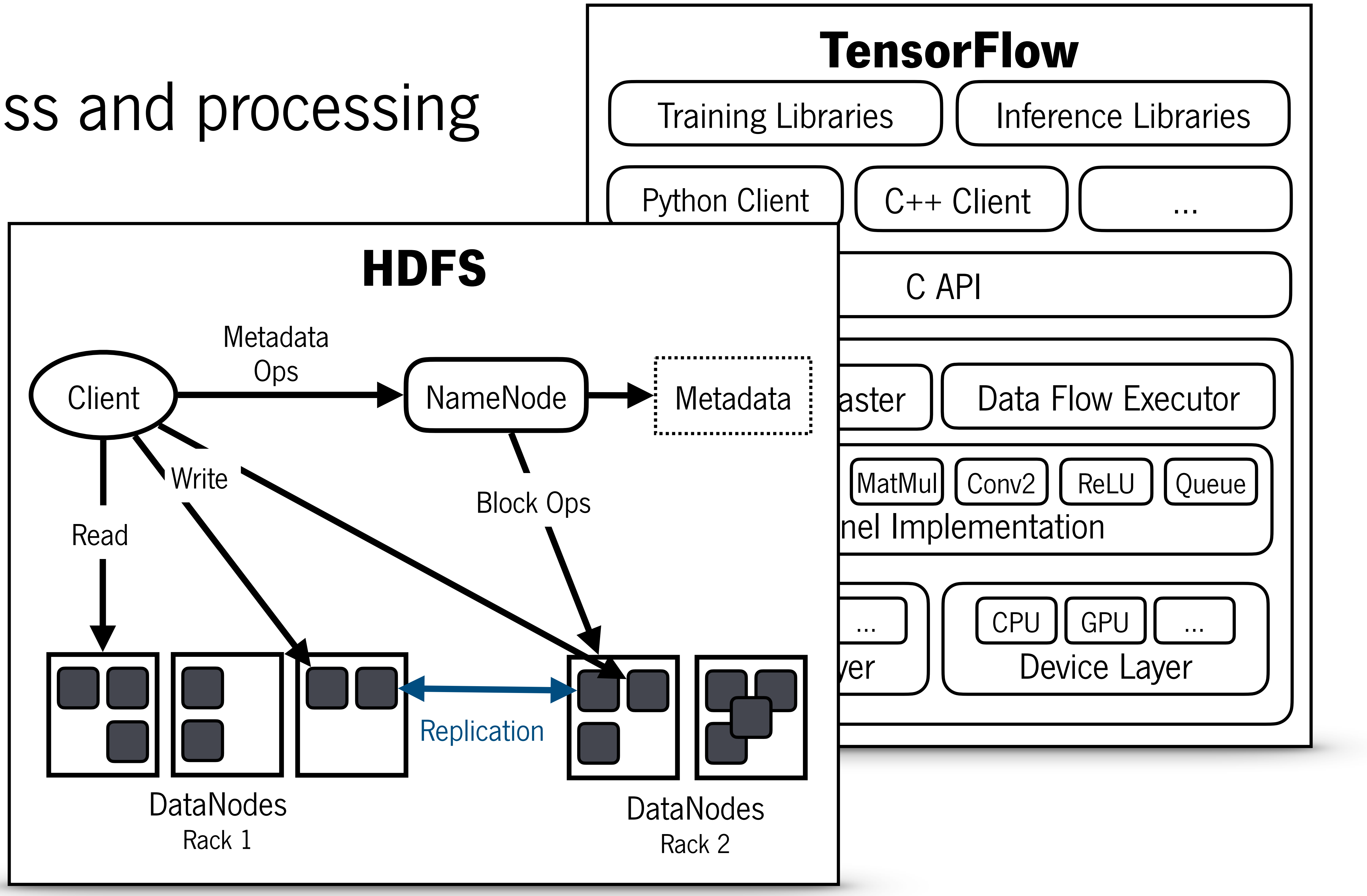
Healthcare

Financial Services

Retail

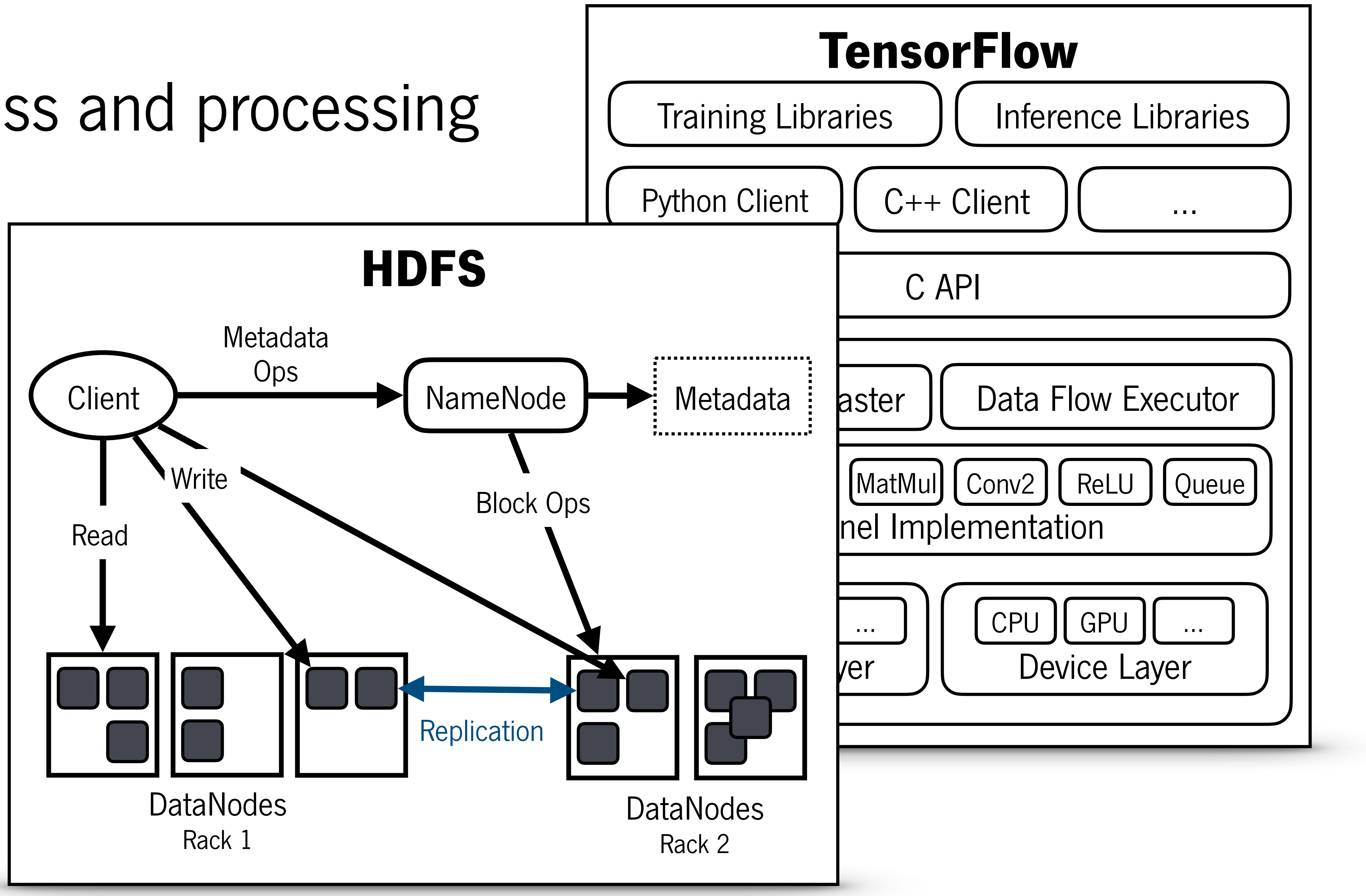
Data-centric and Distributed Systems

- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)



Data-centric and Distributed Systems

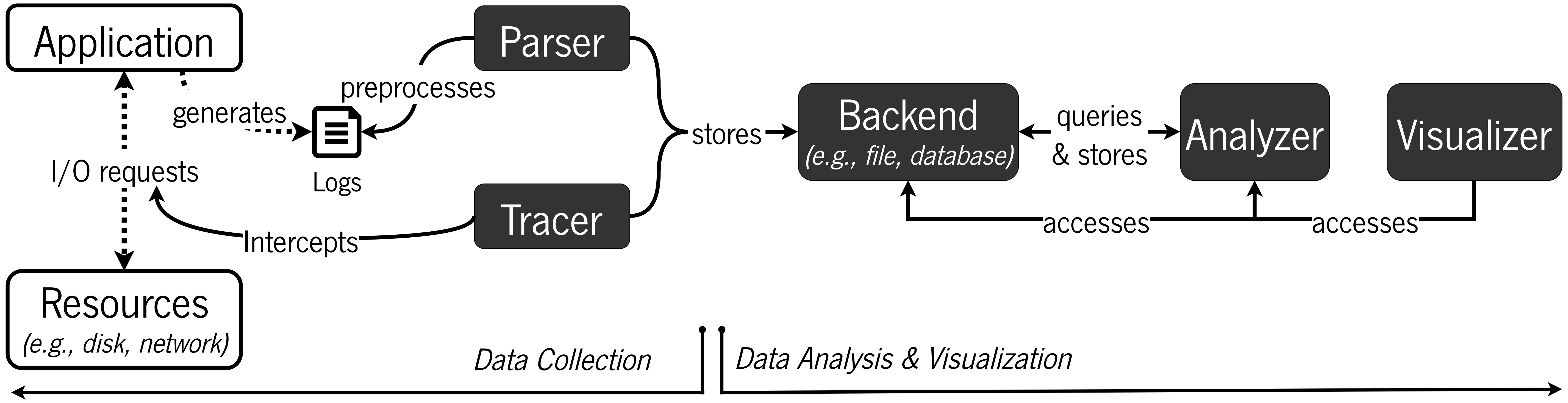
- Critical services increasingly rely on efficient data access and processing
- Complex architectures
  - Large codebases
    - Fluent Bit: ≈1M LoC, 5K files, 4 languages
    - TensorFlow: >4M LoC, 20K files, 3K contributors
  - Several components
  - Complex interactions (e.g., replication)



Question: How can we ensure the correctness and good performance of these systems?

Diagnosis Pipelines

- Provide the collection, analysis and visualization of I/O requests made by applications
- Useful for:
  - Debugging - uncover the root cause of errors, inefficiencies and unattained performance
  - Validation - validate applications' expected behaviors and the corrections of errors
  - Exploration - understand how applications and storage systems handle data requests



Challenges

Challenges

- **#1 - Transparency**
  - Provide transparent solutions that do not require modifications to applications' source code and are generally applicable

Challenges

- **#1 - Transparency**
  - Provide transparent solutions that do not require modifications to applications' source code and are generally applicable

- **#2 - Accuracy and Overhead**
  - Balance the amount and detail of collected data with the overhead imposed on the targeted system

Challenges

◉ #1 - Transparency

- Provide transparent solutions that do not require modifications to applications' source code and are generally applicable

◉ #2 - Accuracy and Overhead

- Balance the amount and detail of collected data with the overhead imposed on the targeted system

◉ #3 - Analysis and Visualization

- Offer practical, integrated, and automated components for analyzing, correlating, and visualizing I/O requests

Challenges

◉ #1 - Transparency

- Provide transparent solutions that do not require modifications to applications' source code and are generally applicable

◉ #2 - Accuracy and Overhead

- Balance the amount and detail of collected data with the overhead imposed on the targeted system

◉ #3 - Analysis and Visualization

- Offer practical, integrated, and automated components for analyzing, correlating, and visualizing I/O requests

◉ #4 - Scope

- Design comprehensive solutions for diagnosing different kinds of I/O behaviors

Contributions

◉ **Content-aware Diagnosis with CaT**

- Enables the collection and analysis of distributed systems' I/O requests

◉ **Comprehensive and Flexible Diagnosis with DIO**

- Provides customizable and insightful diagnosis of data-centric applications' storage I/O

◉ **Custom and Improved Analysis with CRIBA**

- Offers specialized and automated analysis of cryptographic ransomware I/O behavior

Content-Aware Diagnosis

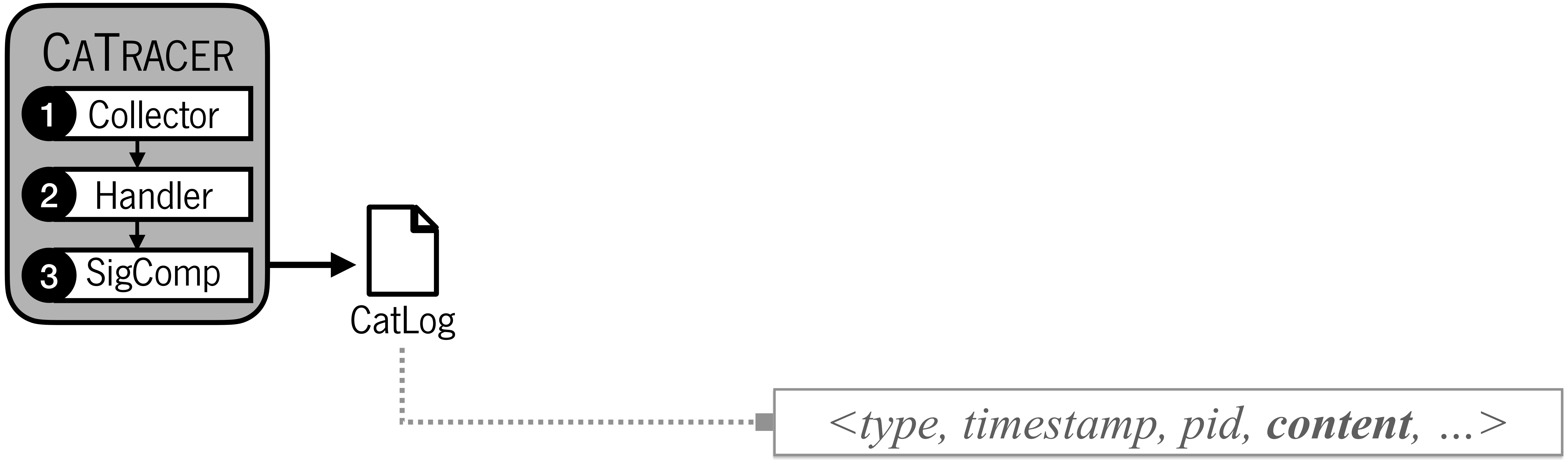
CAT, a framework for diagnosing I/O flow of distributed systems

- Collects requests' context and content
- Combines causality inference with data similarity techniques
- Pinpoints data flow throughout the components of distributed systems

CaT Design

▢ Falcon’s original component ⬚ Falcon’s modified component ▨ New components

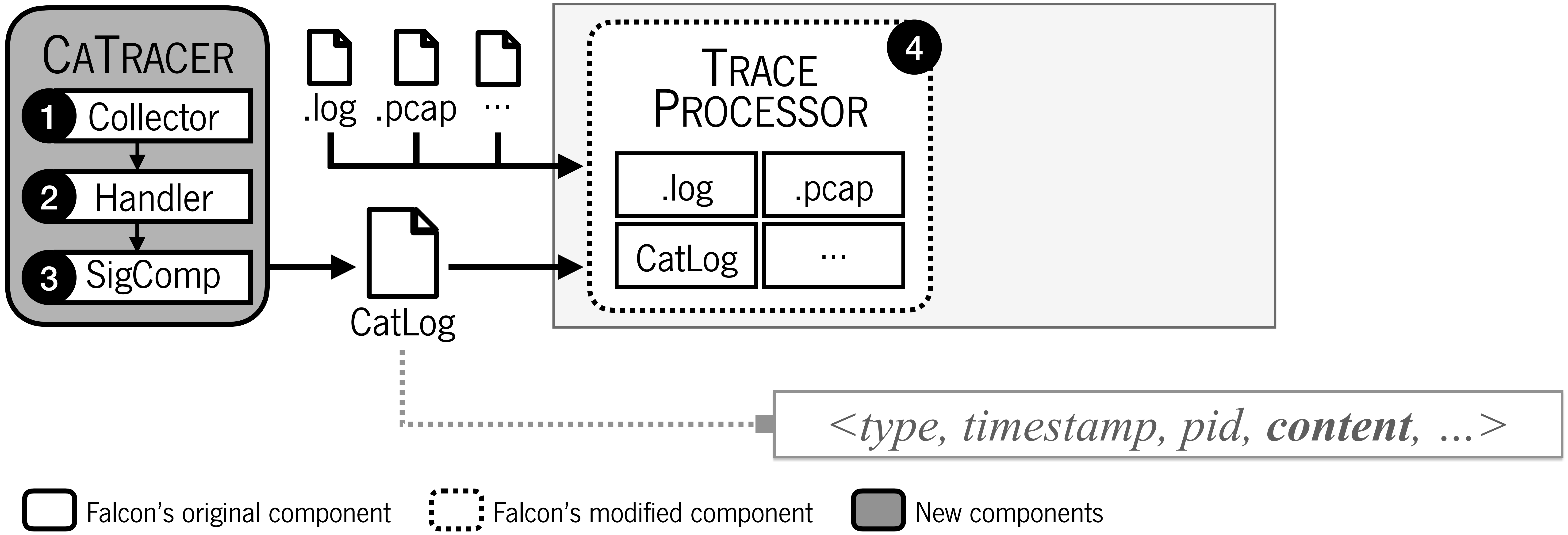
CaT Design



□ Falcon’s original component ⬚ Falcon’s modified component ▨ New components

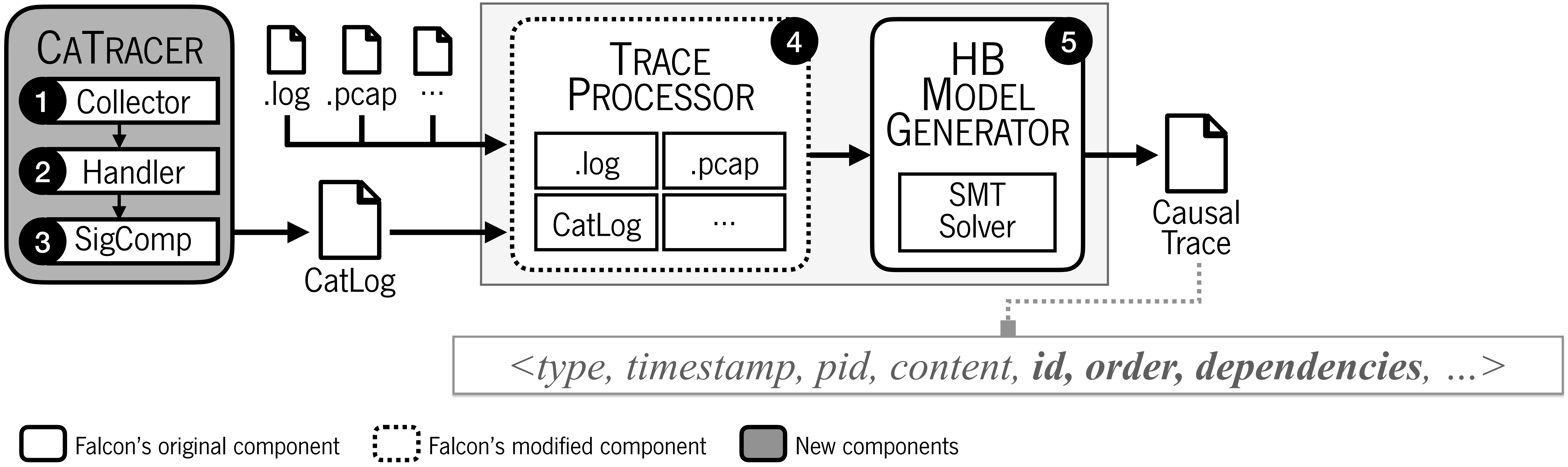
- **CaTracer:** Kernel-level tracer that collects information about I/O requests (events), including their content

CaT Design



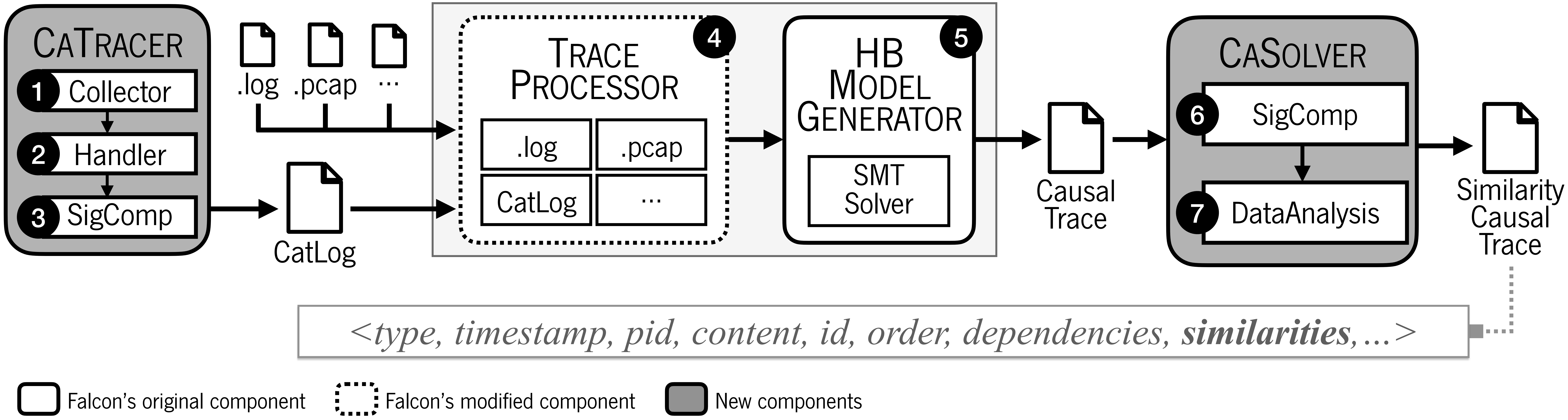
- **CaTracer:** Kernel-level tracer that collects information about I/O requests (events), including their content
- **Trace Processor**: Parses and organizes events into different data structures

CaT Design



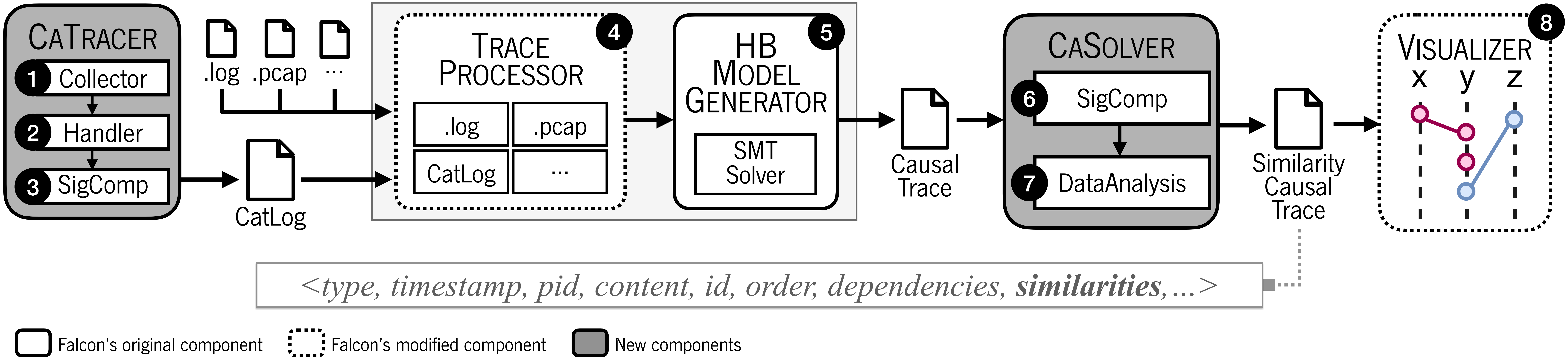
- **CaTracer:** Kernel-level tracer that collects information about I/O requests (events), including their content
- **Trace Processor**: Parses and organizes events into different data structures
- **HB Model Generator**: Infers the causality between events

CaT Design



- ◉**CaTracer:** Kernel-level tracer that collects information about I/O requests (events), including their content
- ◉**Trace Processor**: Parses and organizes events into different data structures
- ◉**HB Model Generator**: Infers the causality between events
- ◉**CaSolver**: Finds events with a high probability of operating over the same data flow

CaT Design

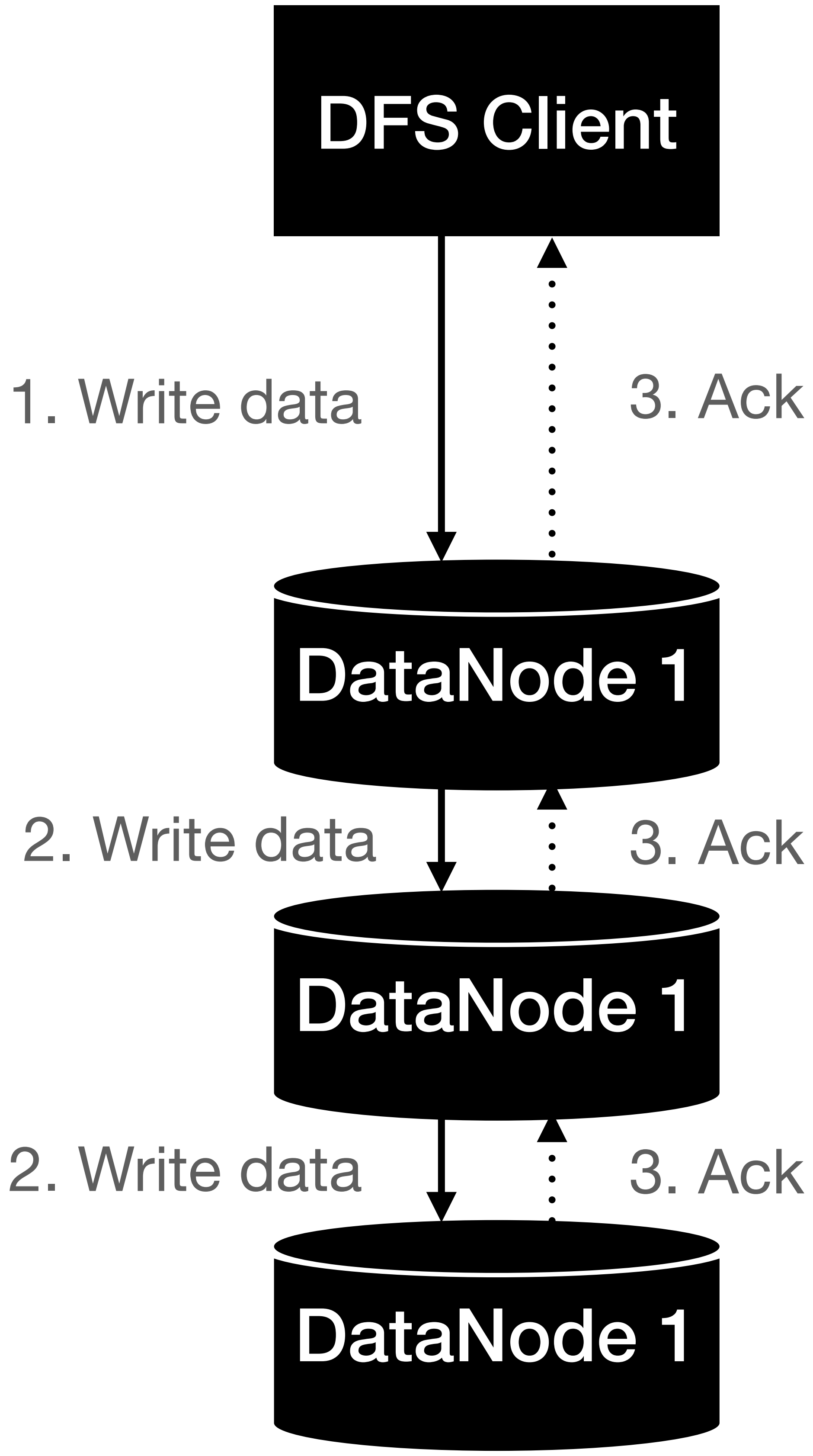


- **CaTracer:** Kernel-level tracer that collects information about I/O requests (events), including their content
- **Trace Processor**: Parses and organizes events into different data structures
- **HB Model Generator**: Infers the causality between events
- **CaSolver**: Finds events with a high probability of operating over the same data flow
- **Visualizer**: Builds space-time diagrams representing the execution, the events' causal relationship and their data flow

CaT in Action

Storage and Replication of a file in HDFS

- **HDFS:** Hadoop distributed file system composed of several DataNodes
- **Replication Process:**
  - Clients send file(s) to one DataNode
  - DataNodes forward data to other nodes and then persist it on disk
  - The process is repeated until all DataNodes have the clients' data
- **3 Test Scenarios:**
  - <u>Normal execution</u>
  - <u>Storage corruption</u>: data modified before being persisted
  - <u>Network corruption</u>: data modified before being transmitted

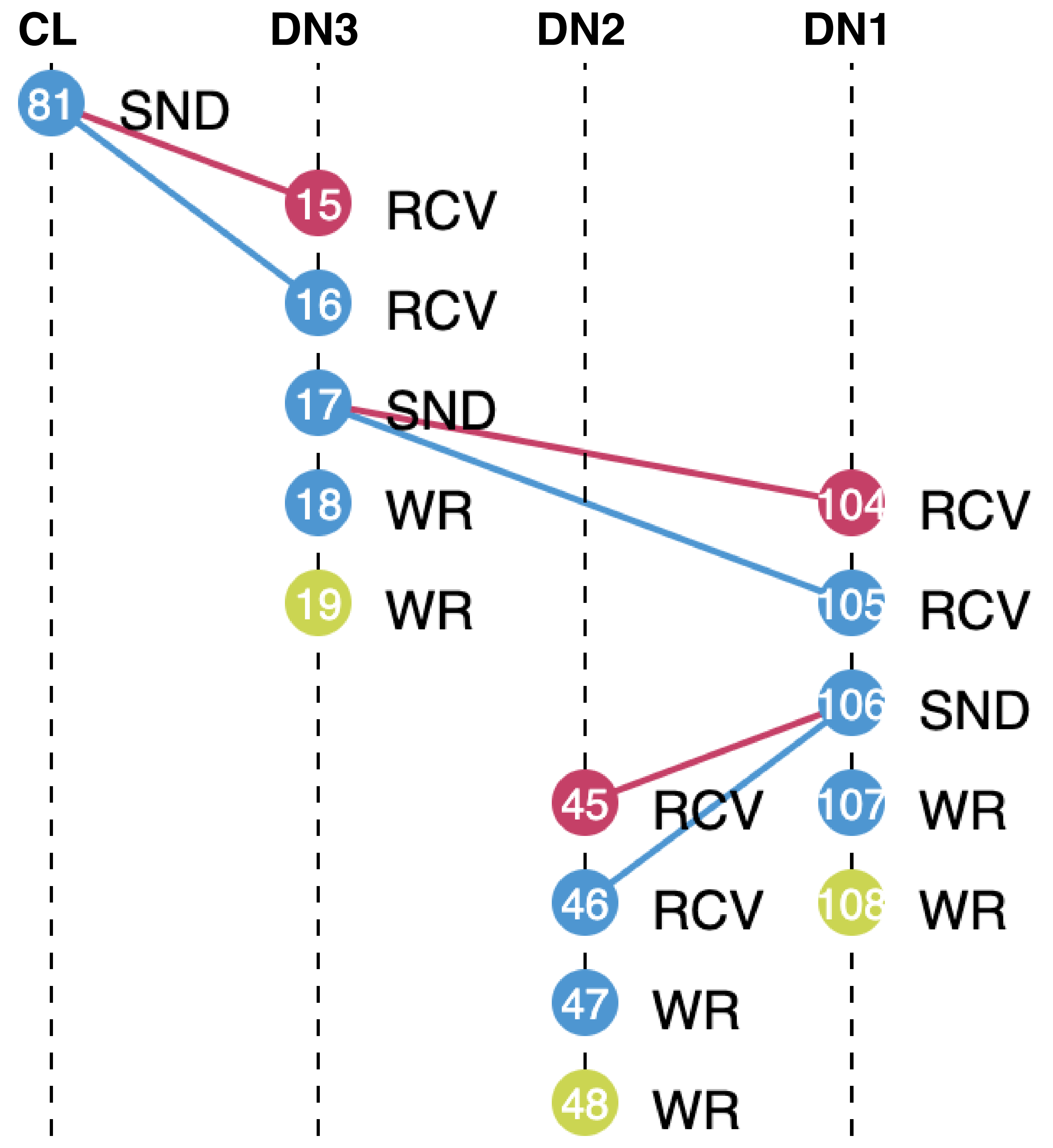


CAT in Action

Storage and Replication of a file in HDFS

CaT in Action

Storage and Replication of a file in HDFS

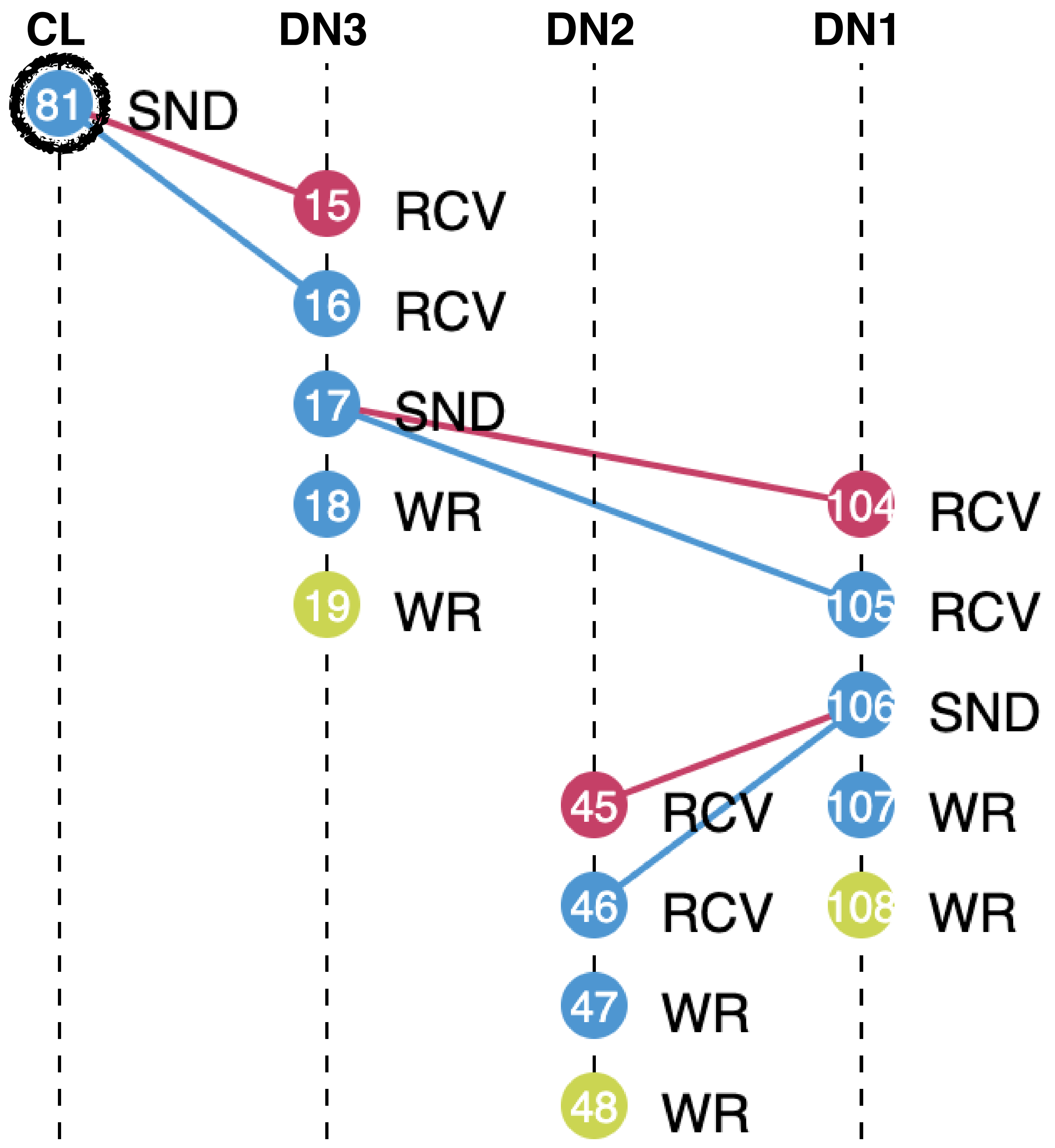


a) Normal execution

All DataNodes persist client's data

CAT in Action

Storage and Replication of a file in HDFS

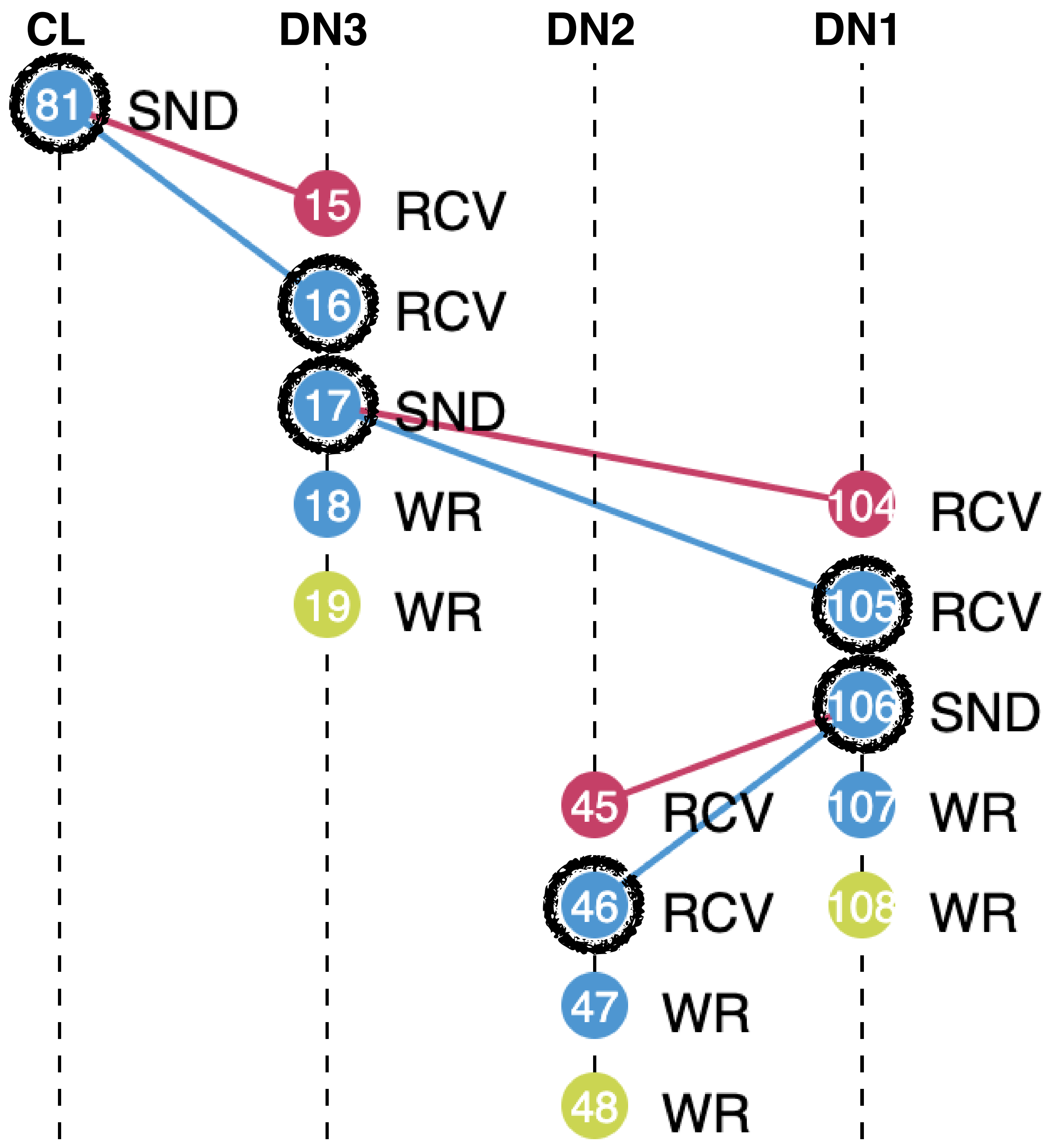


a) Normal execution

All DataNodes persist client's data

CAT in Action

Storage and Replication of a file in HDFS

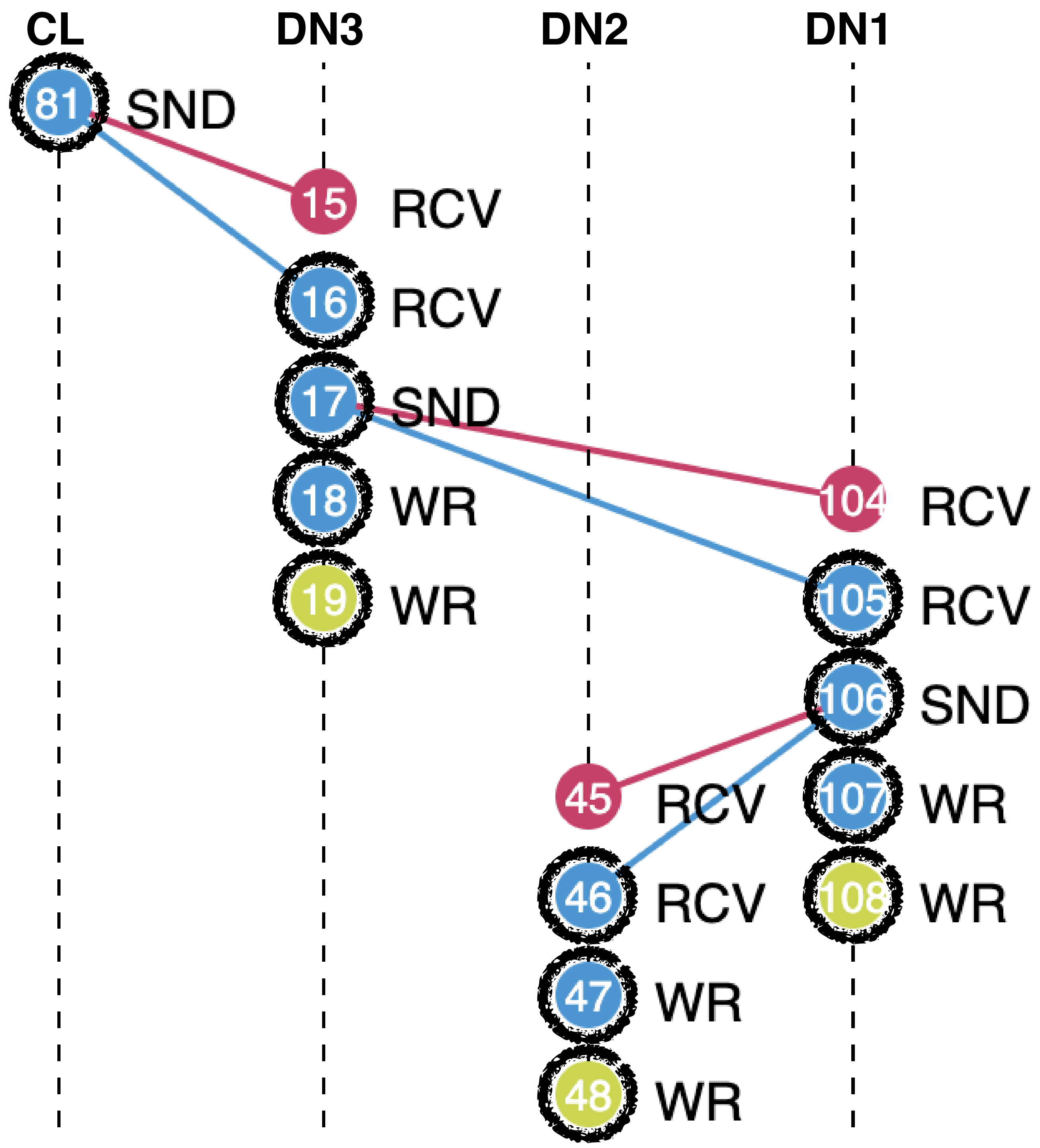


a) Normal execution

All DataNodes persist client's data

CaT in Action

Storage and Replication of a file in HDFS

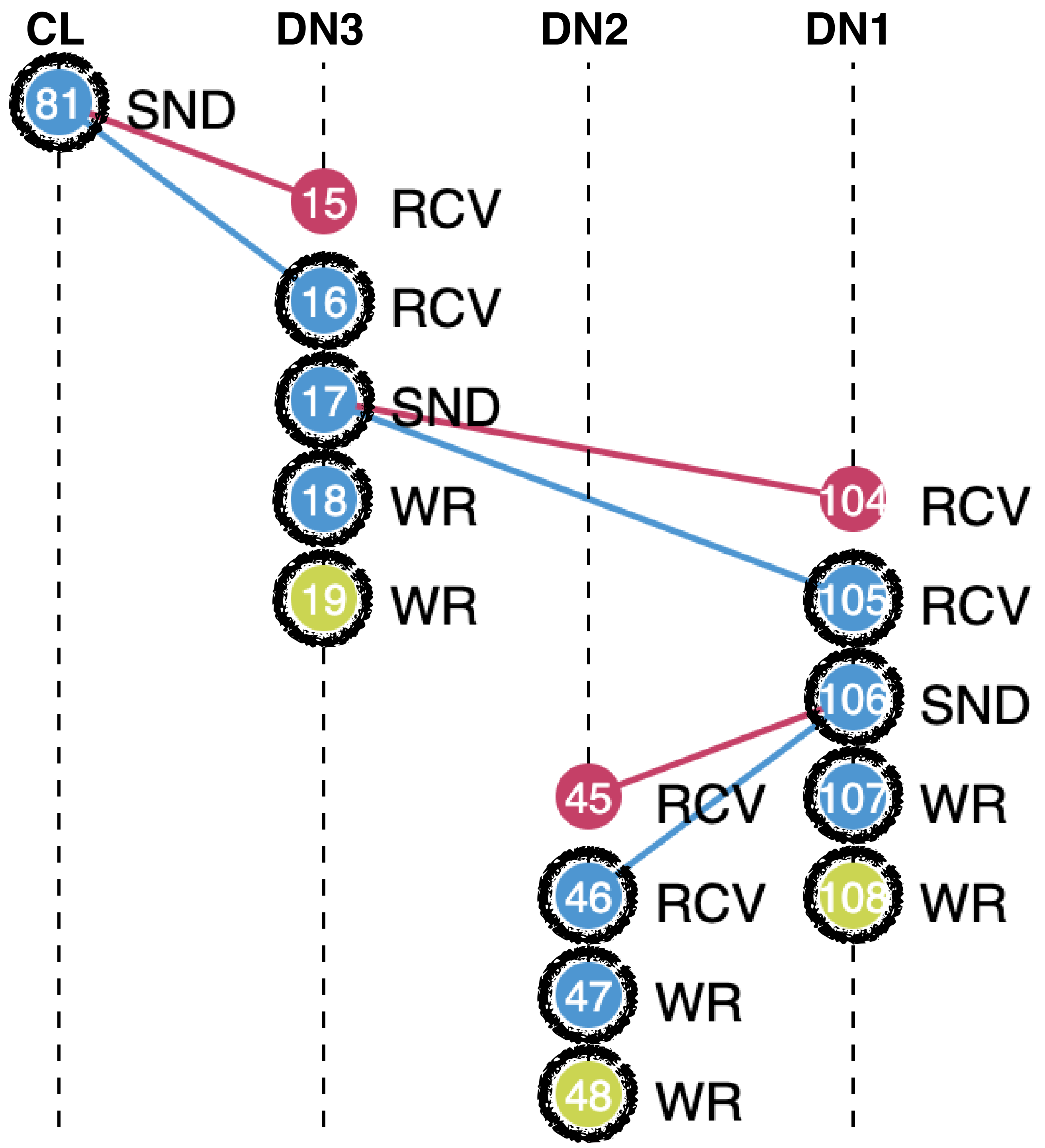


a) Normal execution

All DataNodes persist client's data

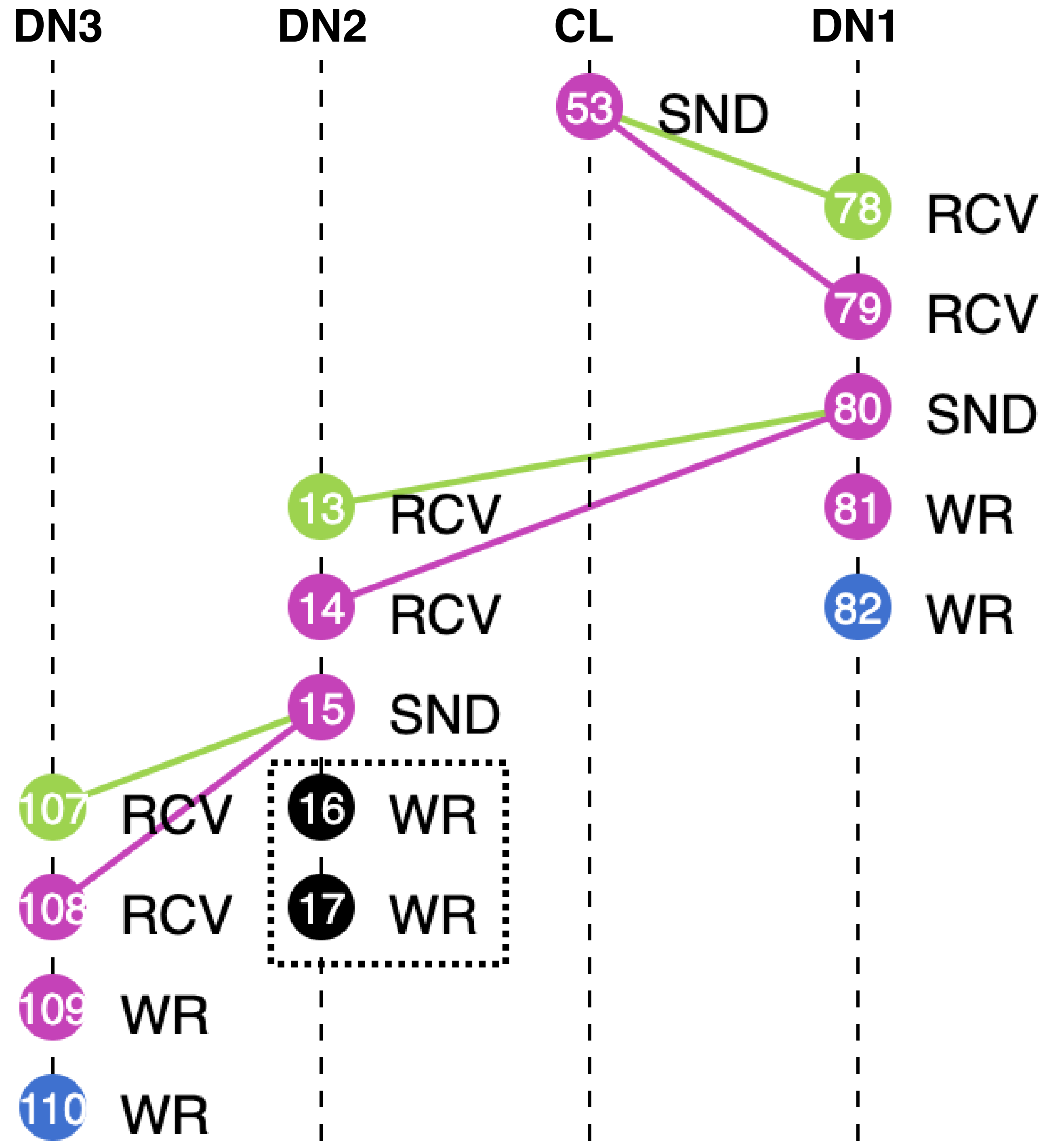
CaT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

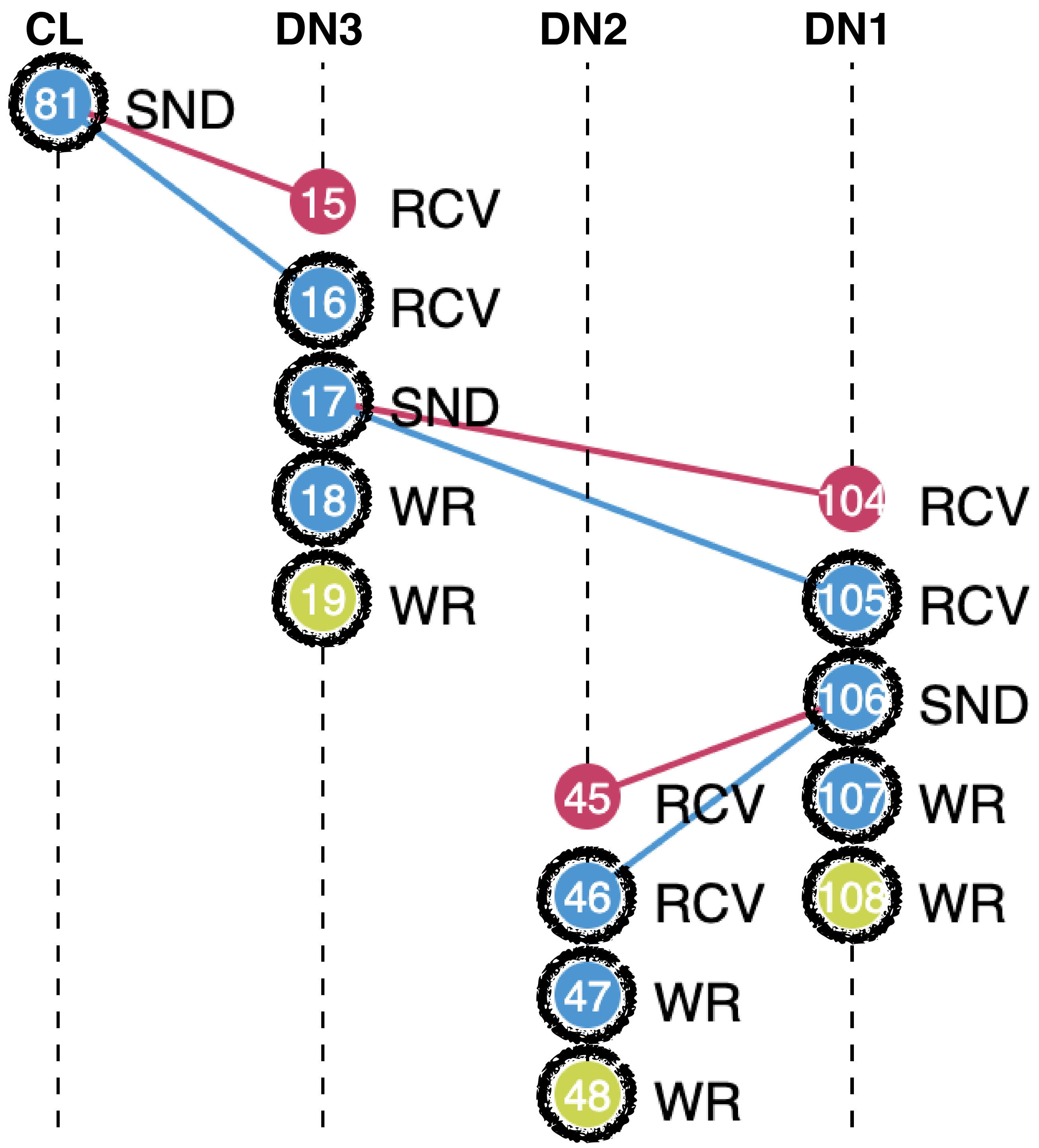


b) Storage corruption

Data persisted by DataNode 2 differs from clients' data

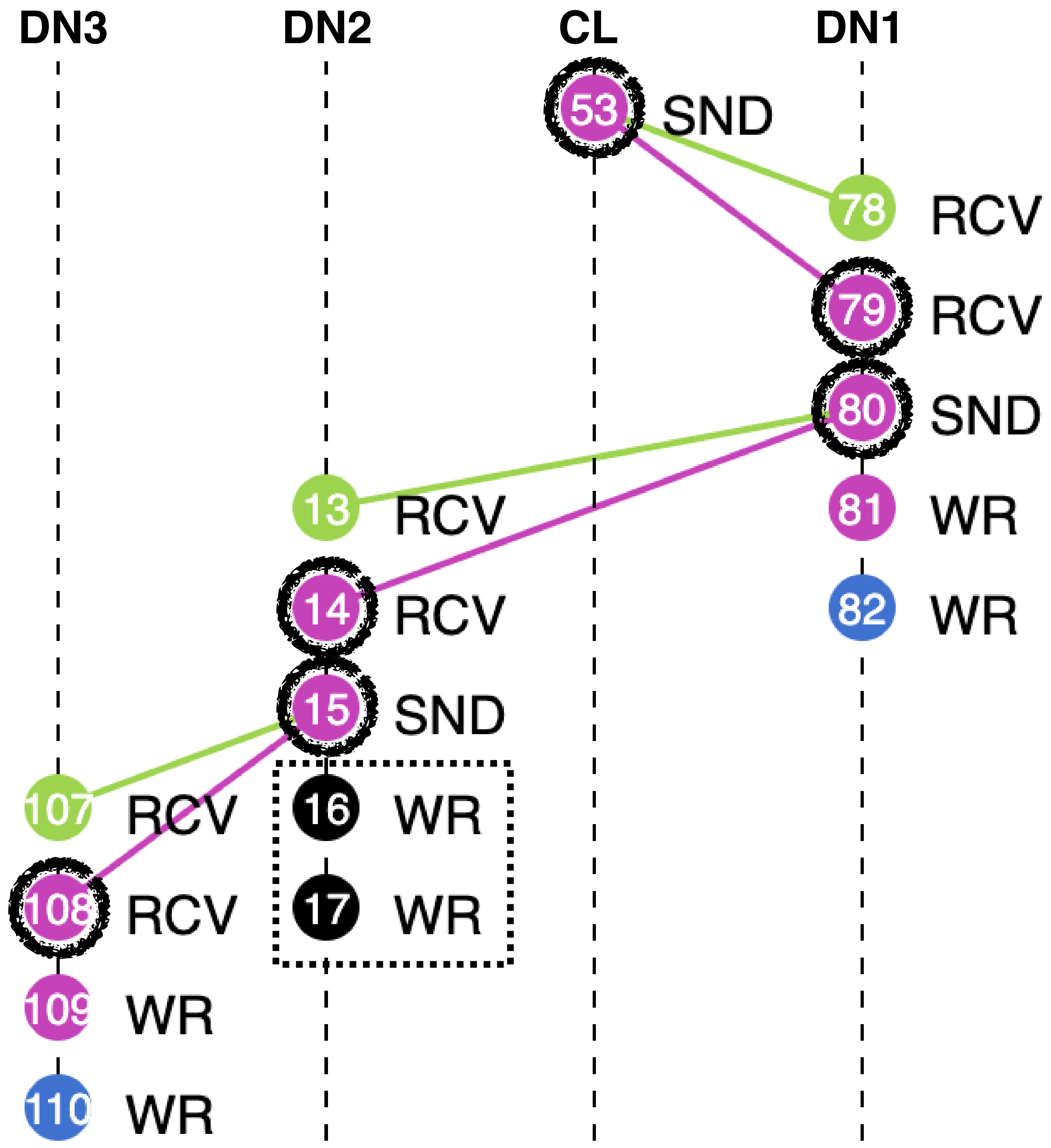
CaT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

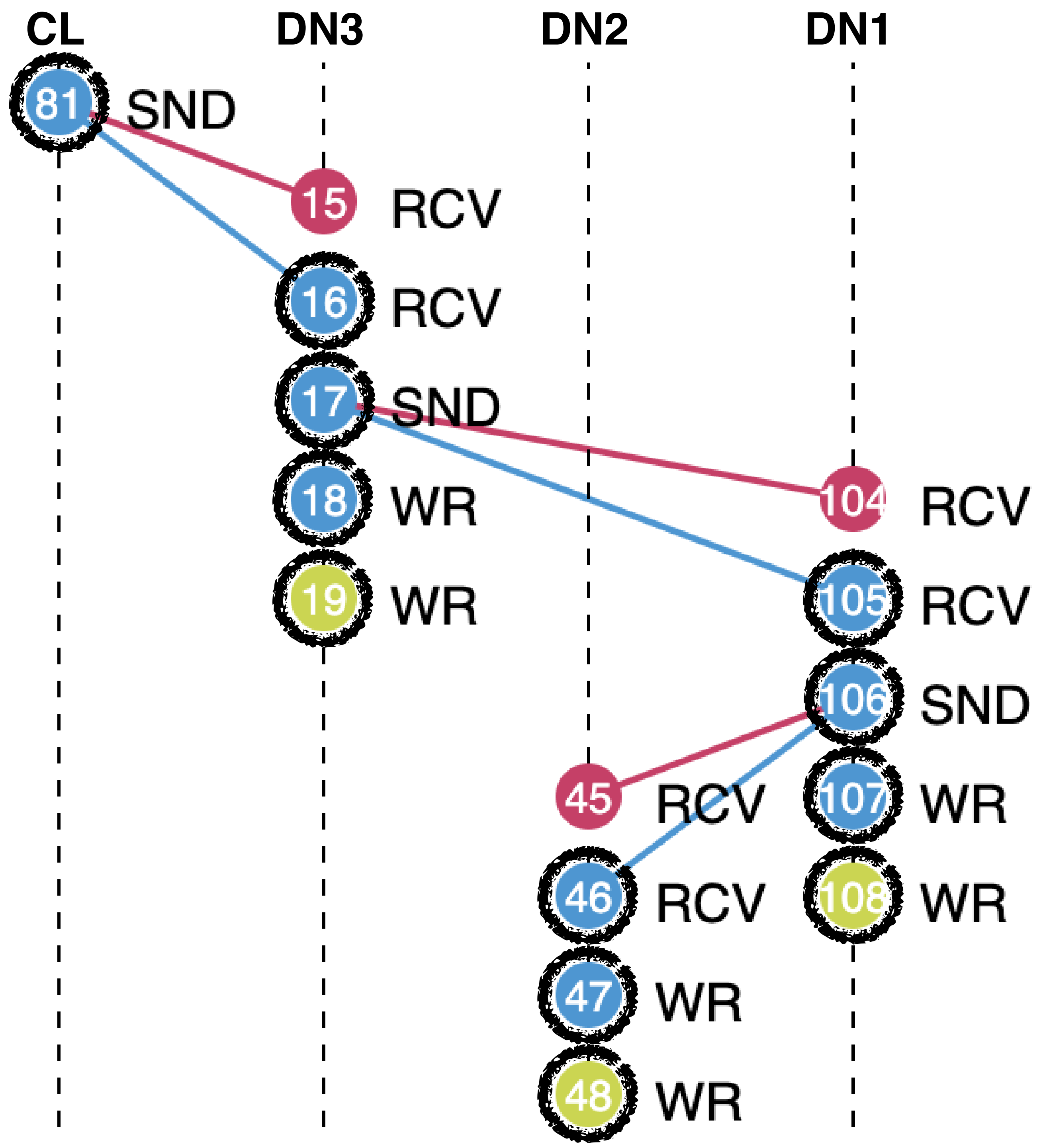


b) Storage corruption

Data persisted by DataNode 2 differs from clients' data

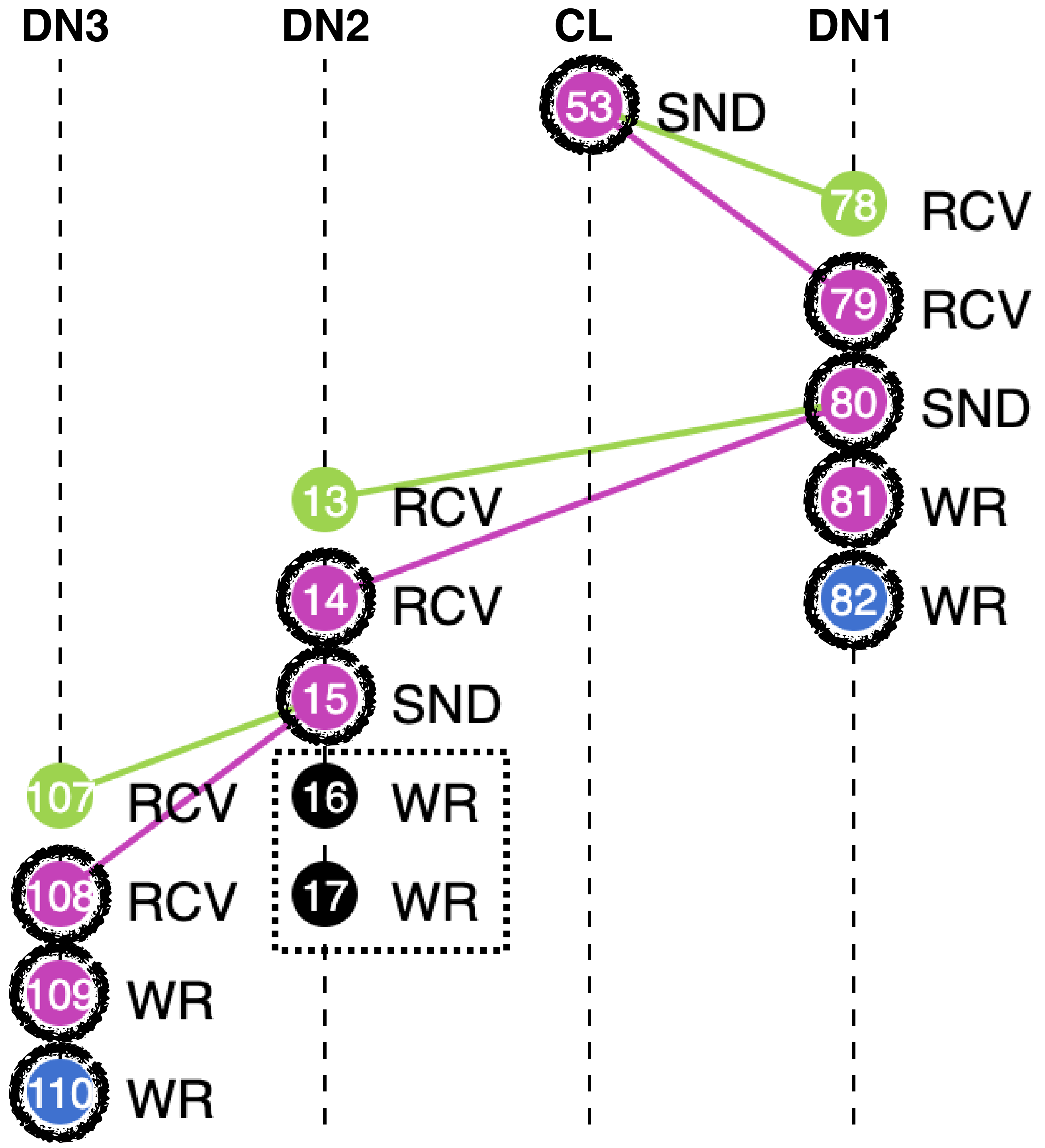
CaT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

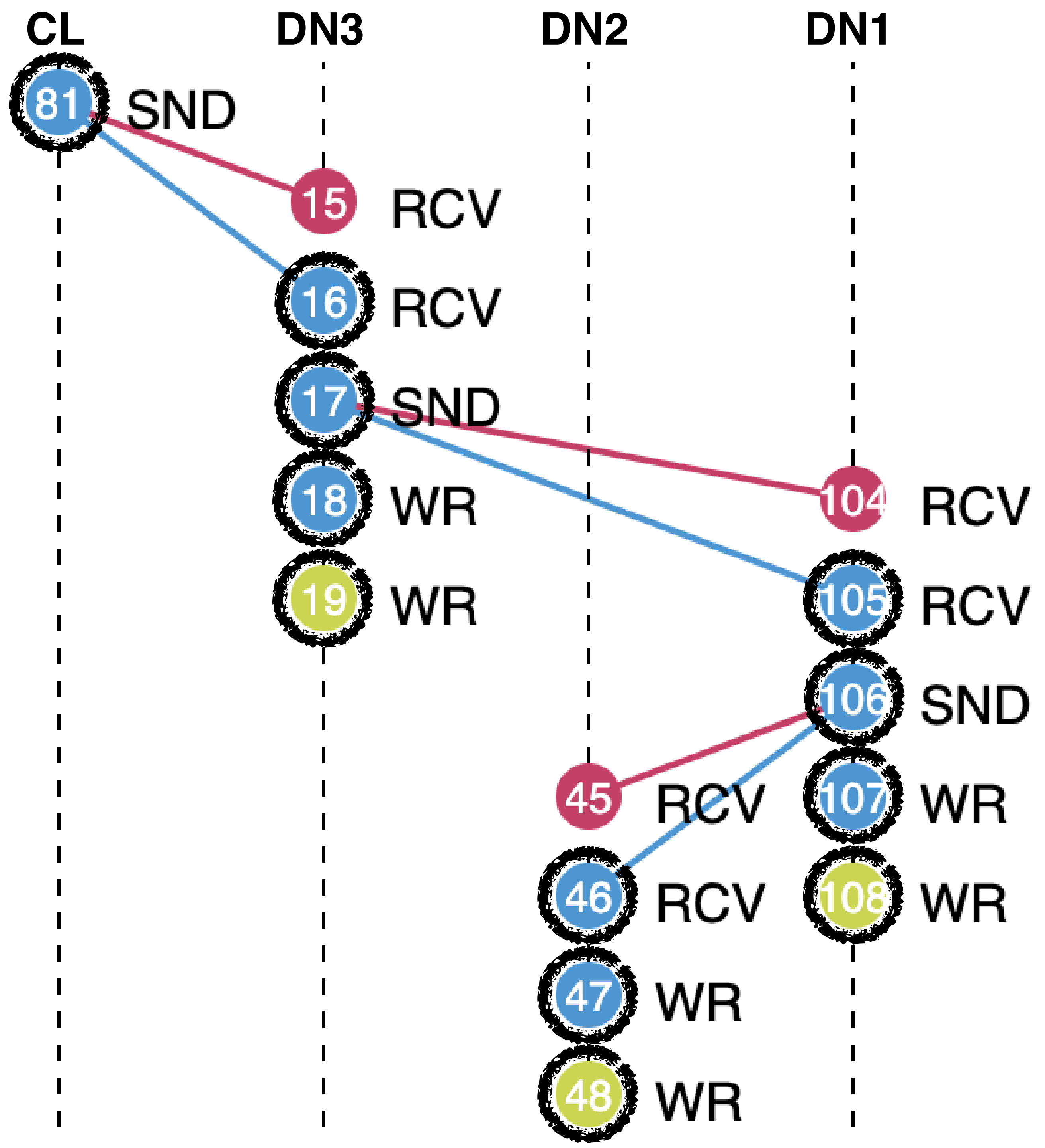


b) Storage corruption

Data persisted by DataNode 2 differs from clients' data

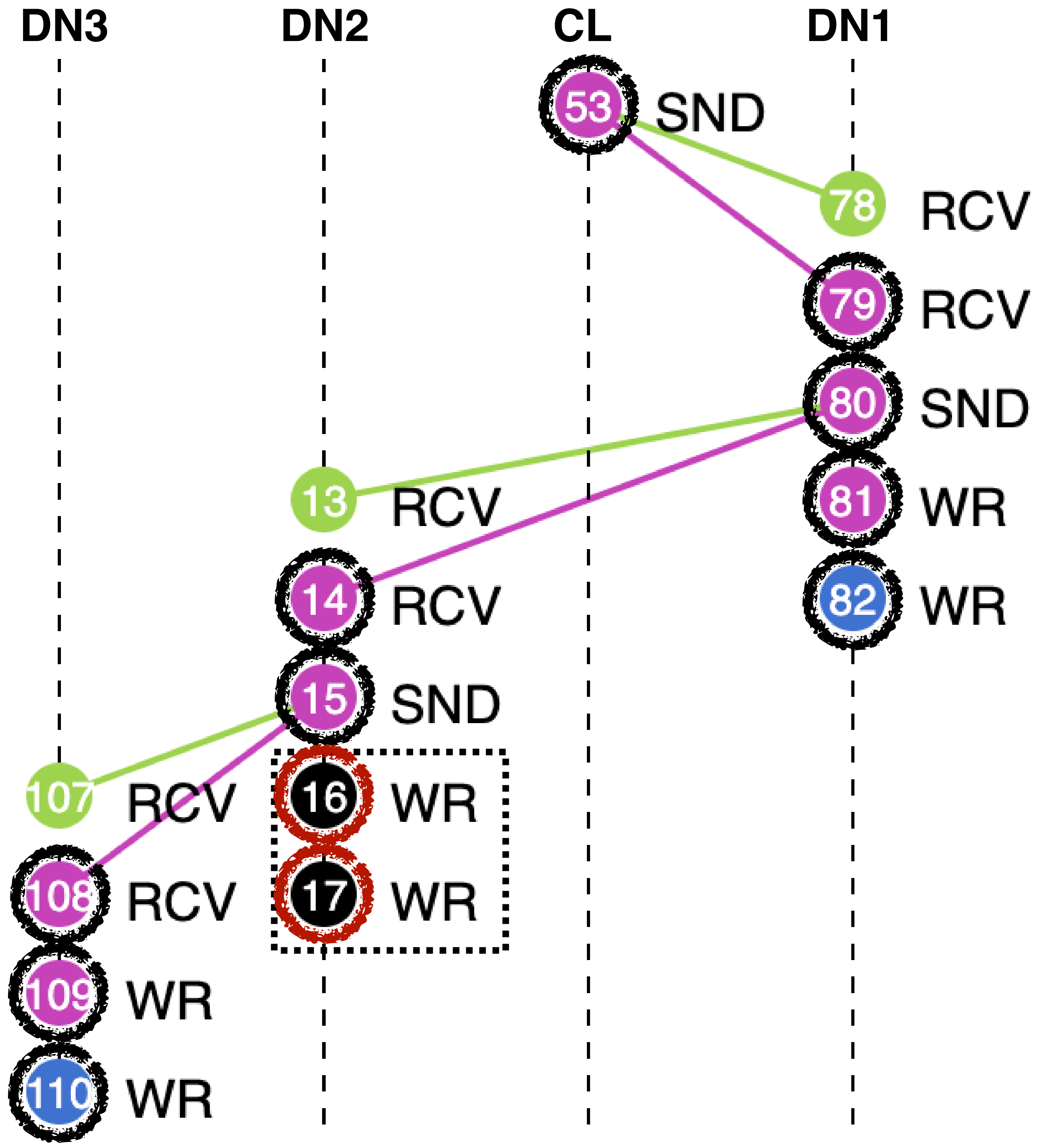
CAT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

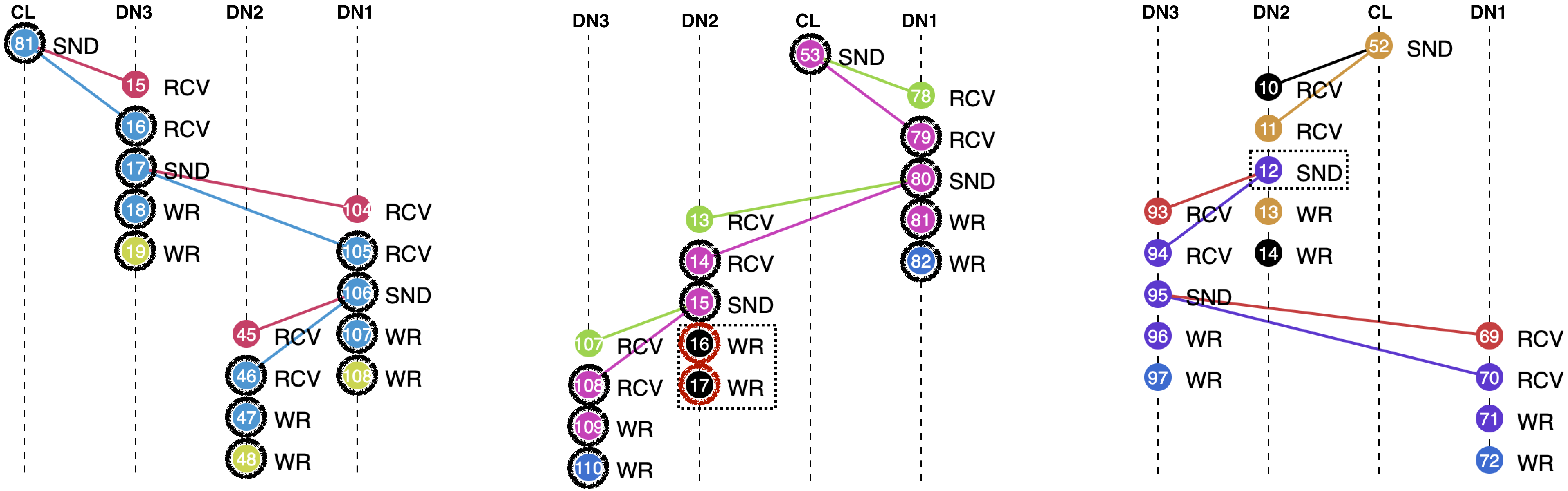


b) Storage corruption

Data persisted by DataNode 2 differs from clients' data

CaT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

b) Storage corruption

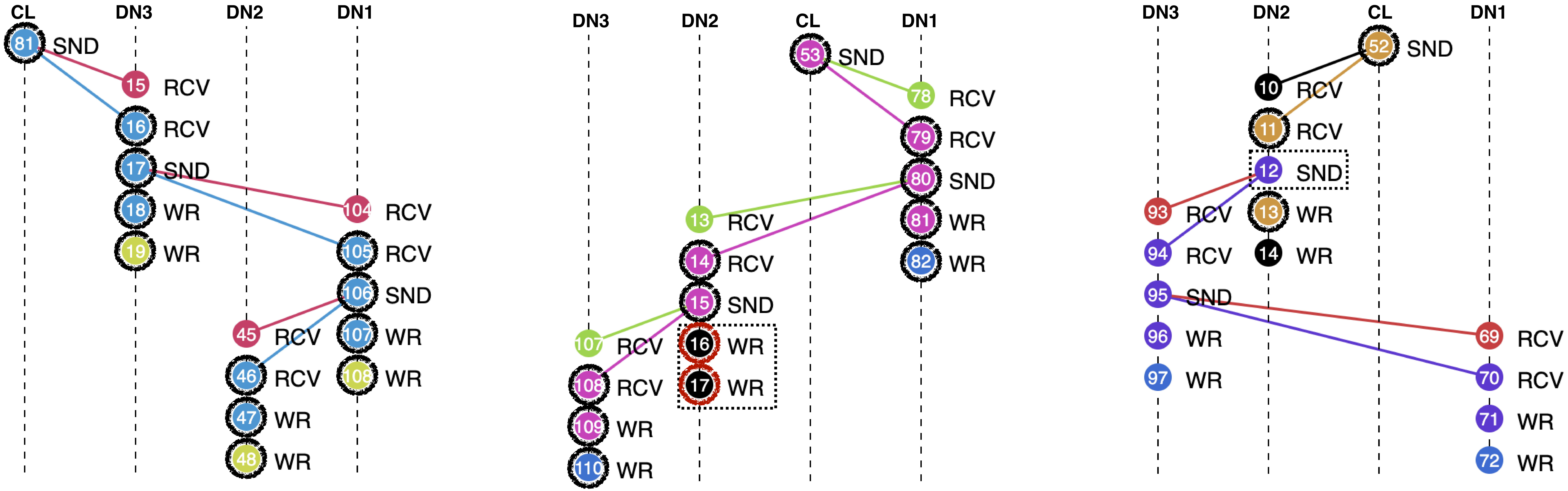
Data persisted by DataNode 2 differs from clients' data

c) Network corruption

Data transmitted by DataNode 2 differs from clients' data

CaT in Action

Storage and Replication of a file in HDFS



a) Normal execution

All DataNodes persist client's data

b) Storage corruption

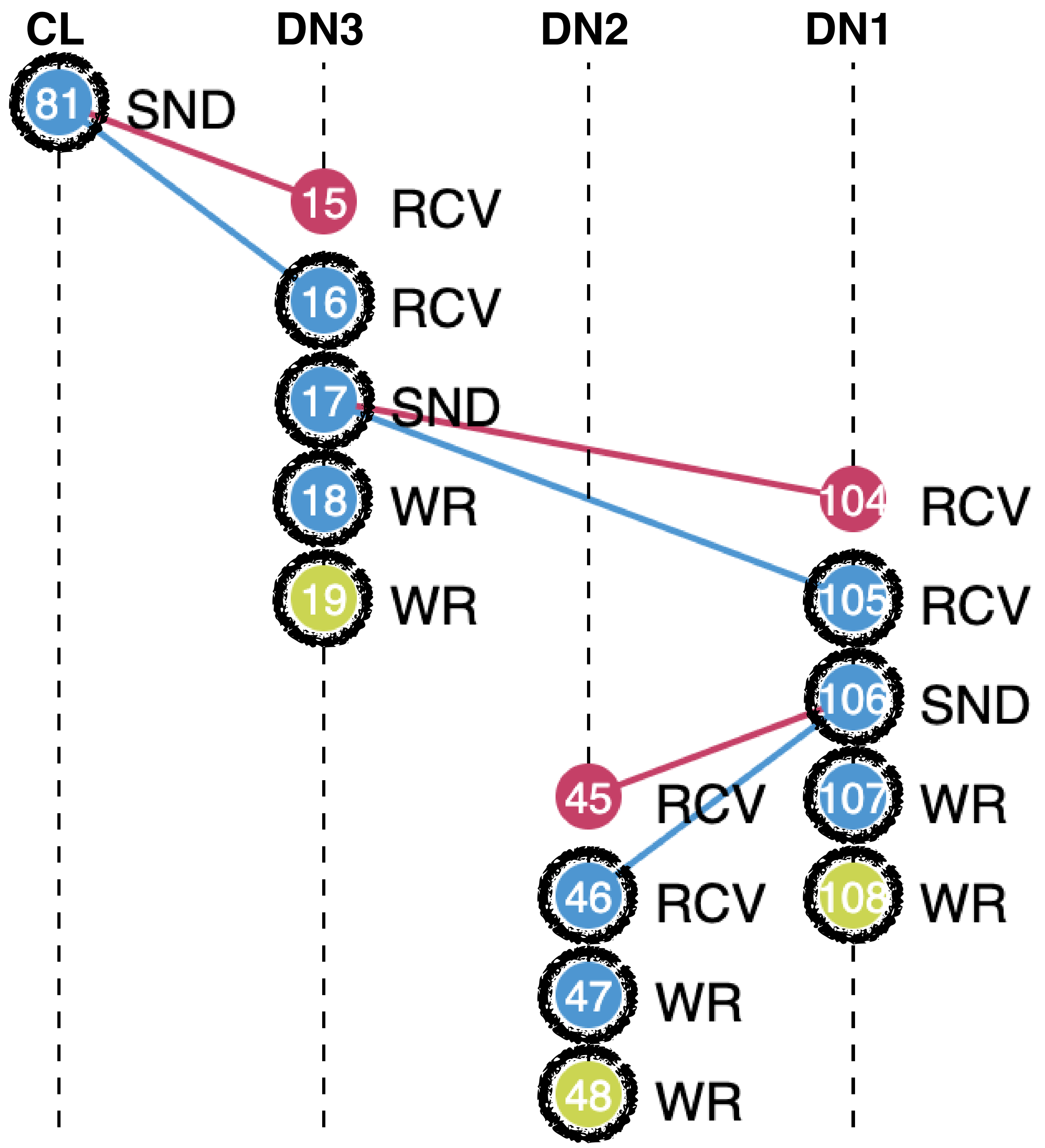
Data persisted by DataNode 2 differs from clients' data

c) Network corruption

Data transmitted by DataNode 2 differs from clients' data

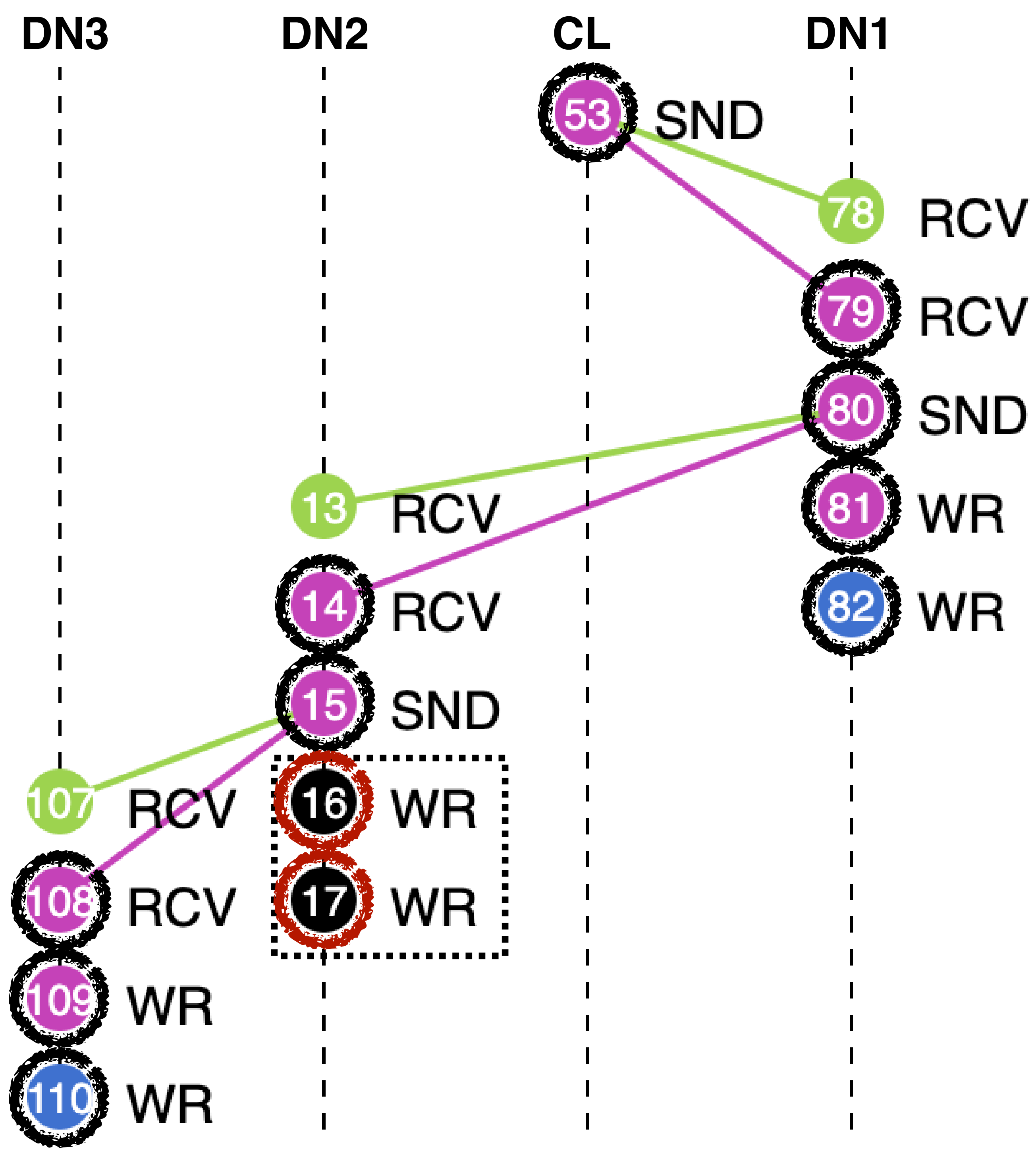
CaT in Action

Storage and Replication of a file in HDFS



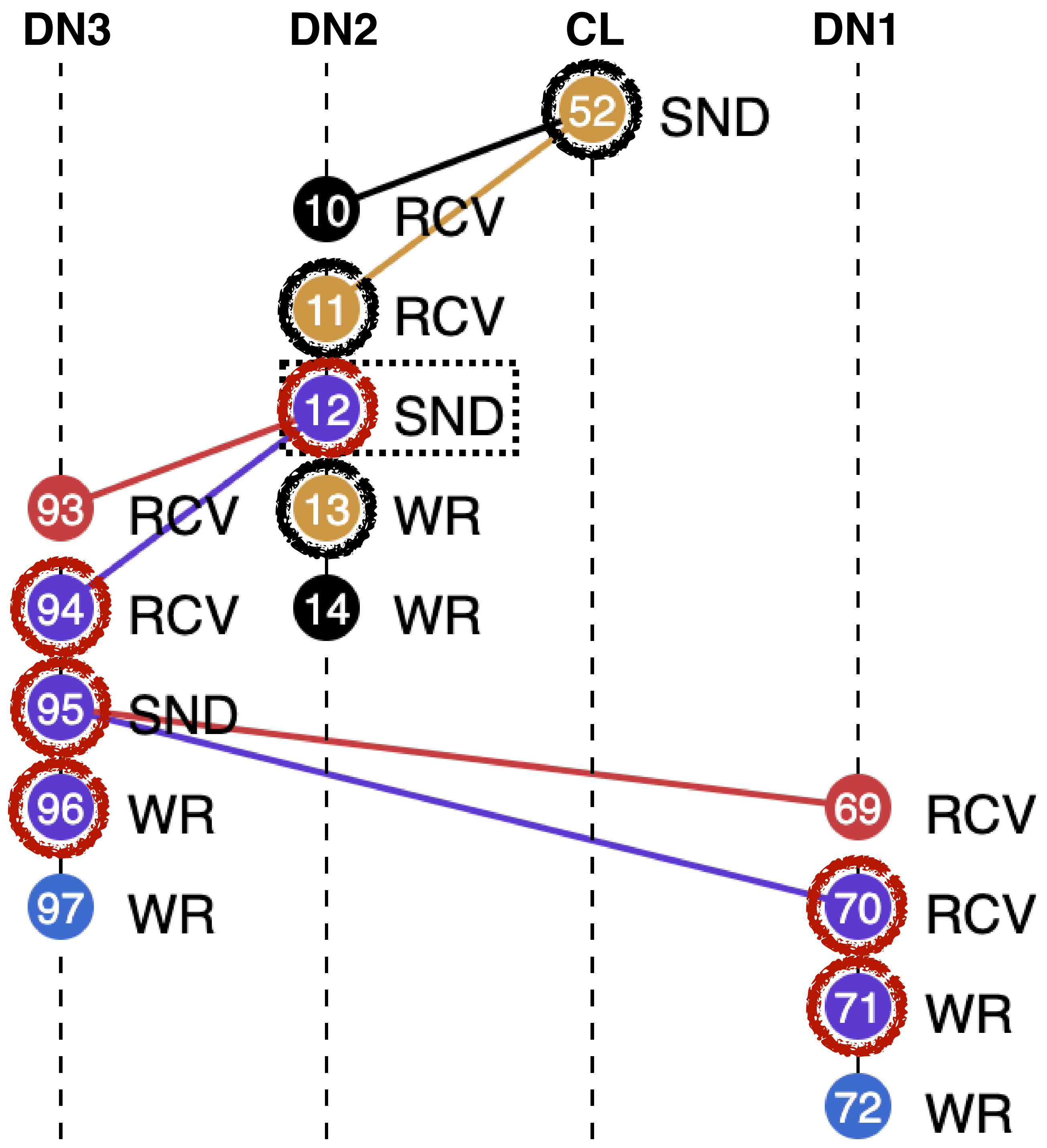
a) Normal execution

All DataNodes persist client's data



b) Storage corruption

Data persisted by DataNode 2 differs from clients' data



c) Network corruption

Data transmitted by DataNode 2 differs from clients' data

CaT Summary

- CaT's content-aware approach enables the detection of **data adulteration, corruption and leakage patterns** that would go **unnoticed with state-of-the-art context-based** solutions

- **Open challenges**:
  - Comprehensive diagnosis of applications
  - Practical and efficient analysis pipeline

Comprehensive and Flexible Diagnosis

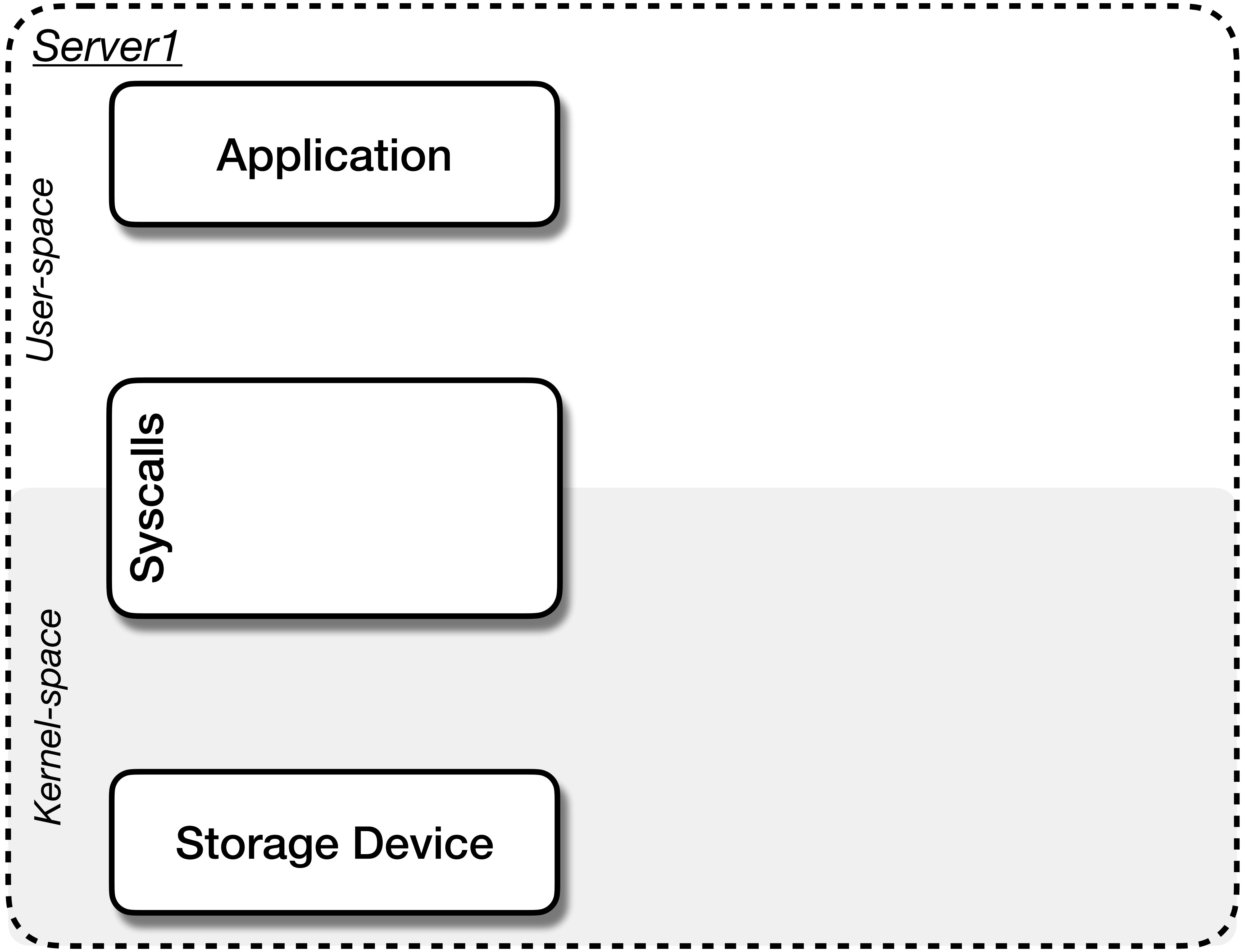
DIO, a generic tool for diagnosing applications’ storage I/O

- Supports 42 storage-related system calls
- Collects their type, arguments, return value and extra context from the kernel
- Provides different strategies to customize the amount and detail of collected data
- Includes an integrated pipeline for near real-time analysis and visualization

DIO Design

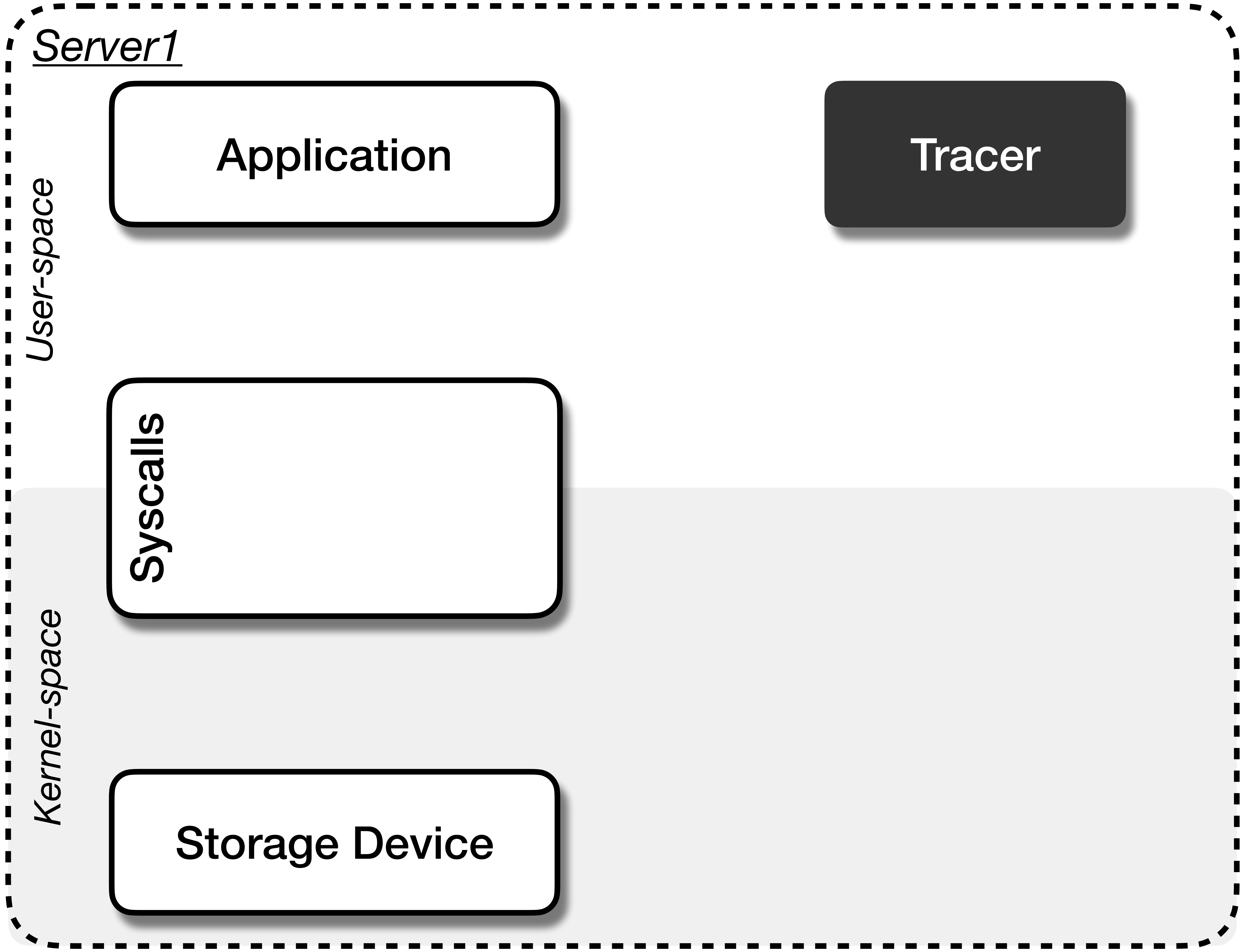
■ DIO’s components → DIO main flow ⇢ App flow

DIO Design



■ DIO's components → DIO main flow - - -▸ App flow

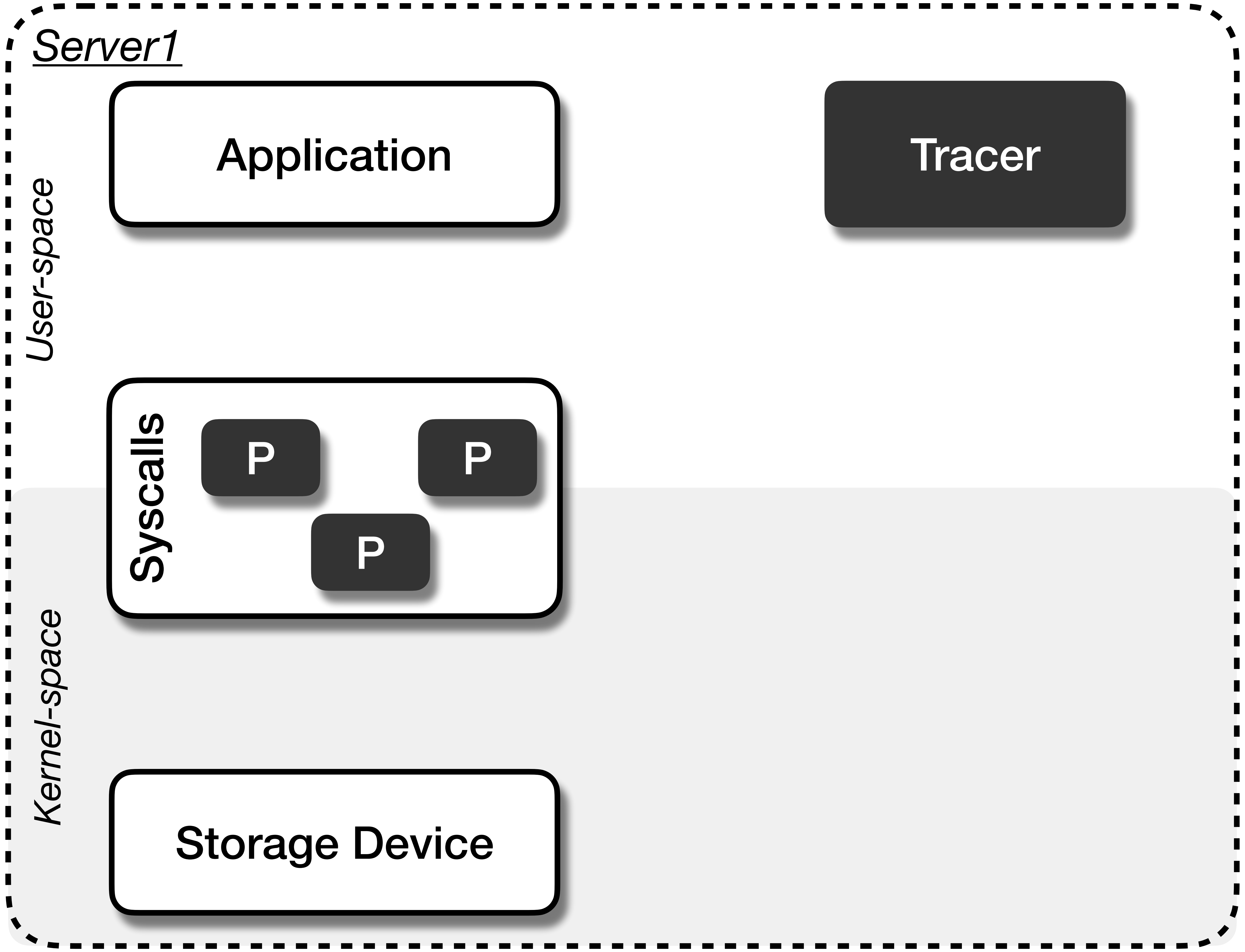
DIO Design



DIO’s tracer runs along the targeted application, intercepting its syscalls

■ DIO’s components → DIO main flow ⇢ App flow

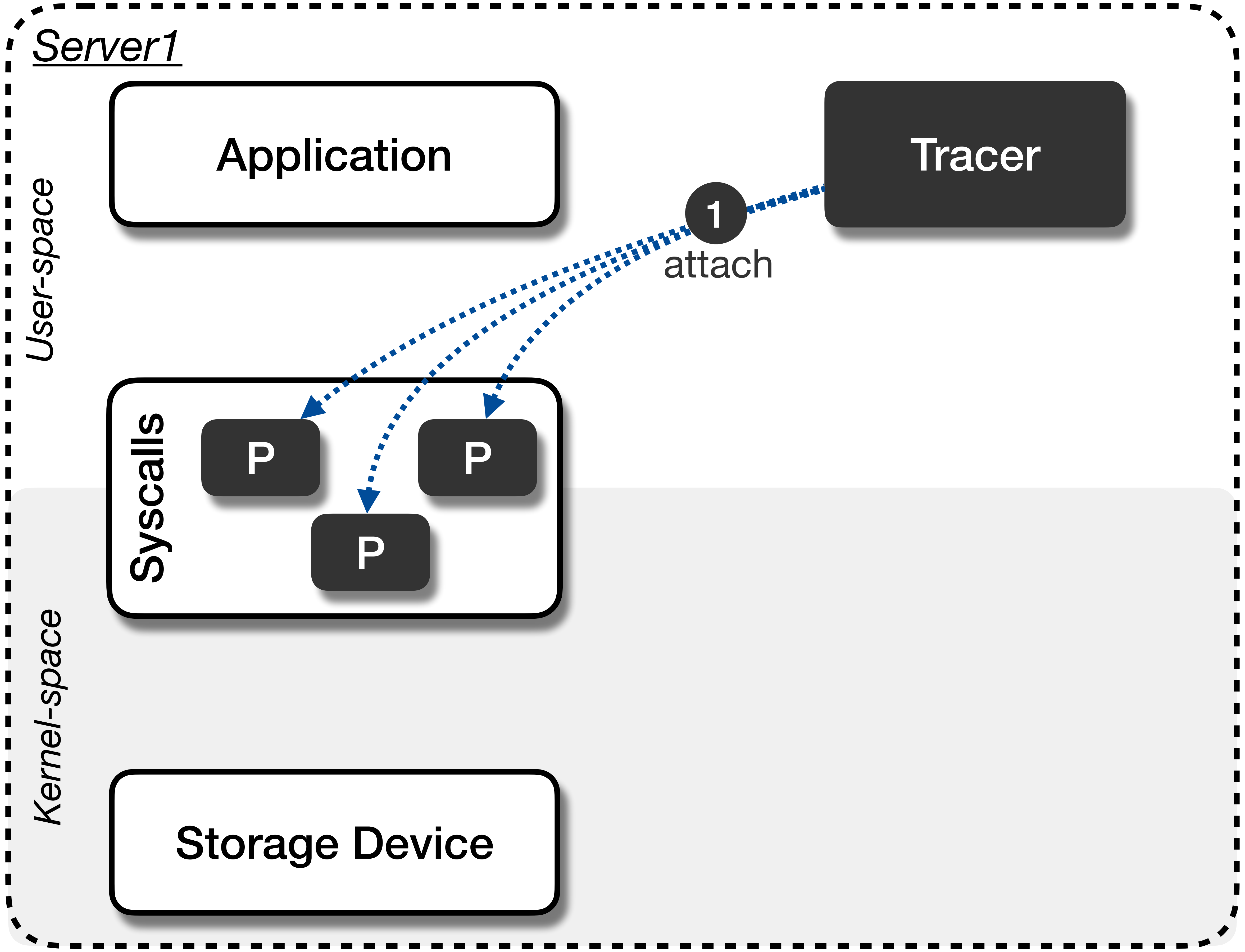
DIO Design



DIO's tracer runs along the targeted application, intercepting its syscalls

■ DIO's components ⟶ DIO main flow - - -▸ App flow

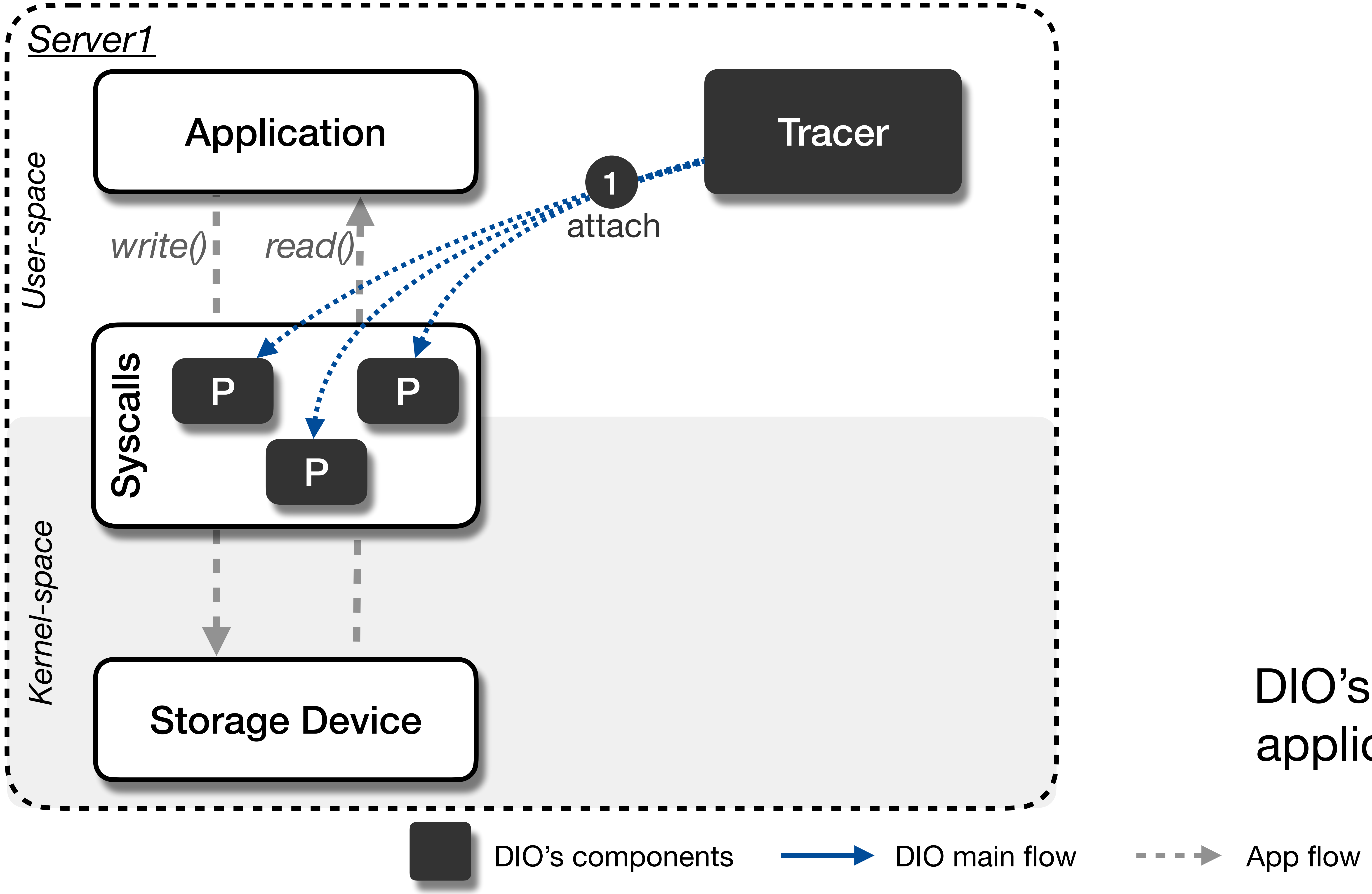
DIO Design



■ DIO's components ⟶ DIO main flow - - -➤ App flow

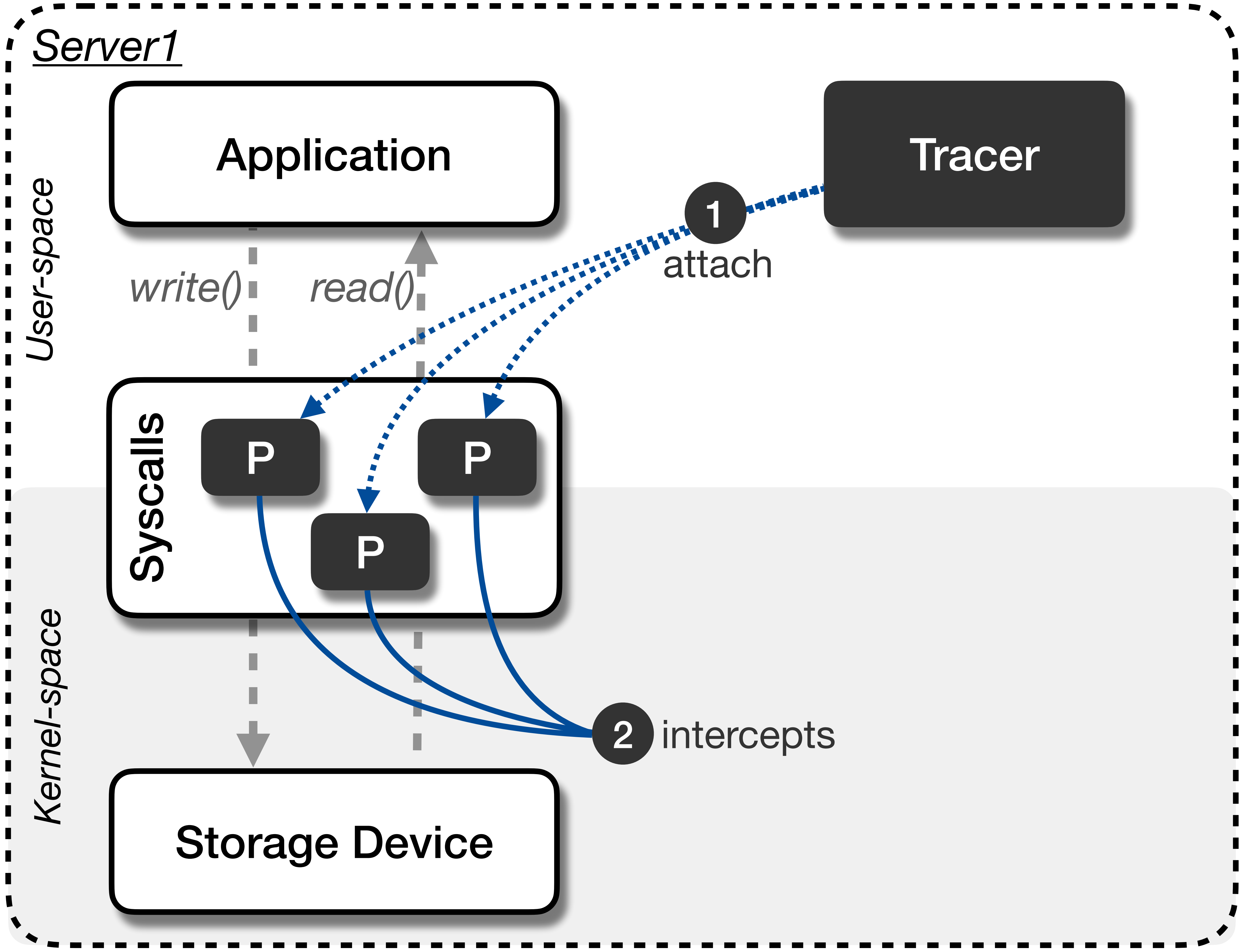
DIO's tracer runs along the targeted application, intercepting its syscalls

DIO Design



DIO's tracer runs along the targeted application, intercepting its syscalls

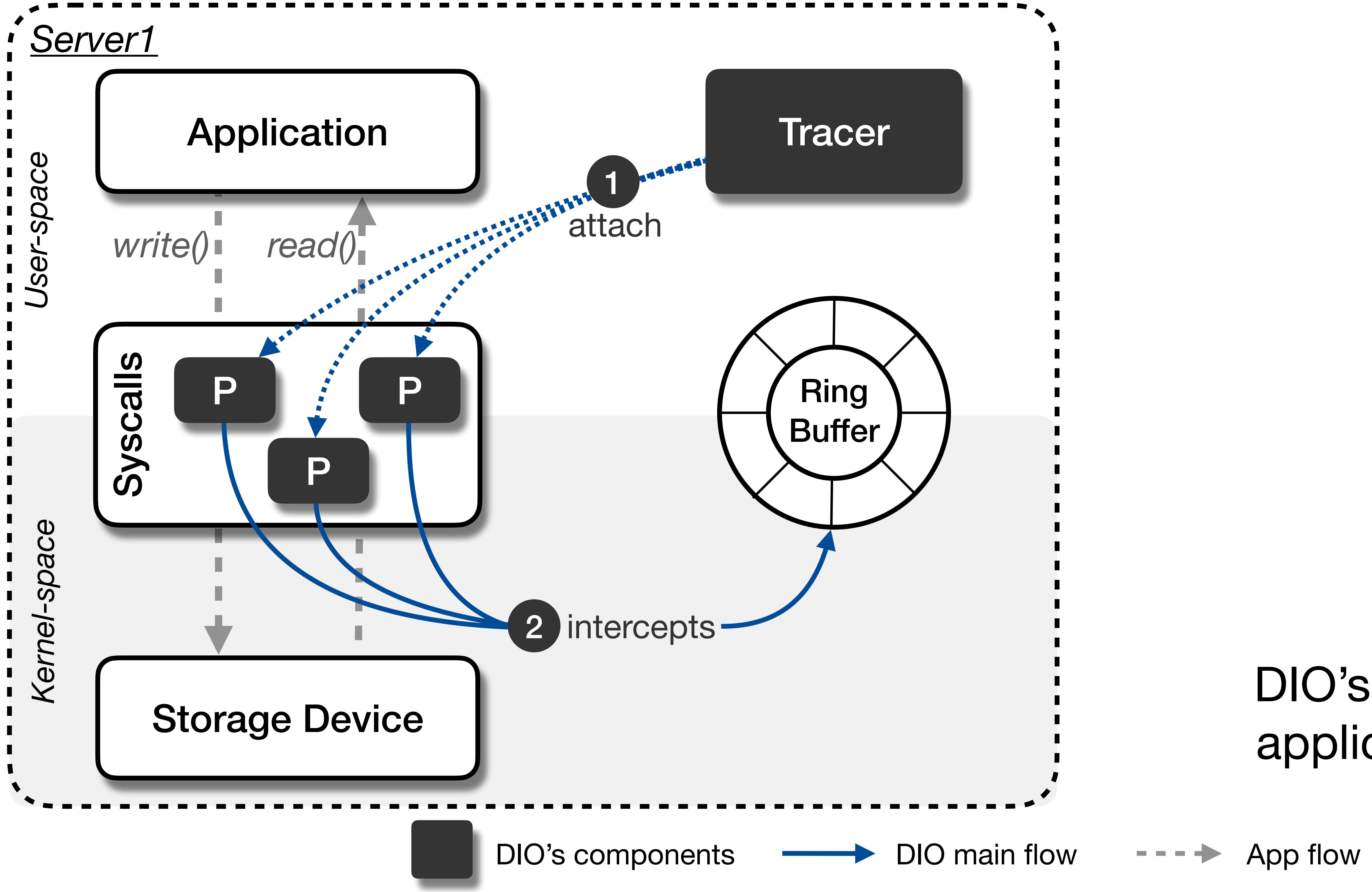
DIO Design



DIO's tracer runs along the targeted application, intercepting its syscalls

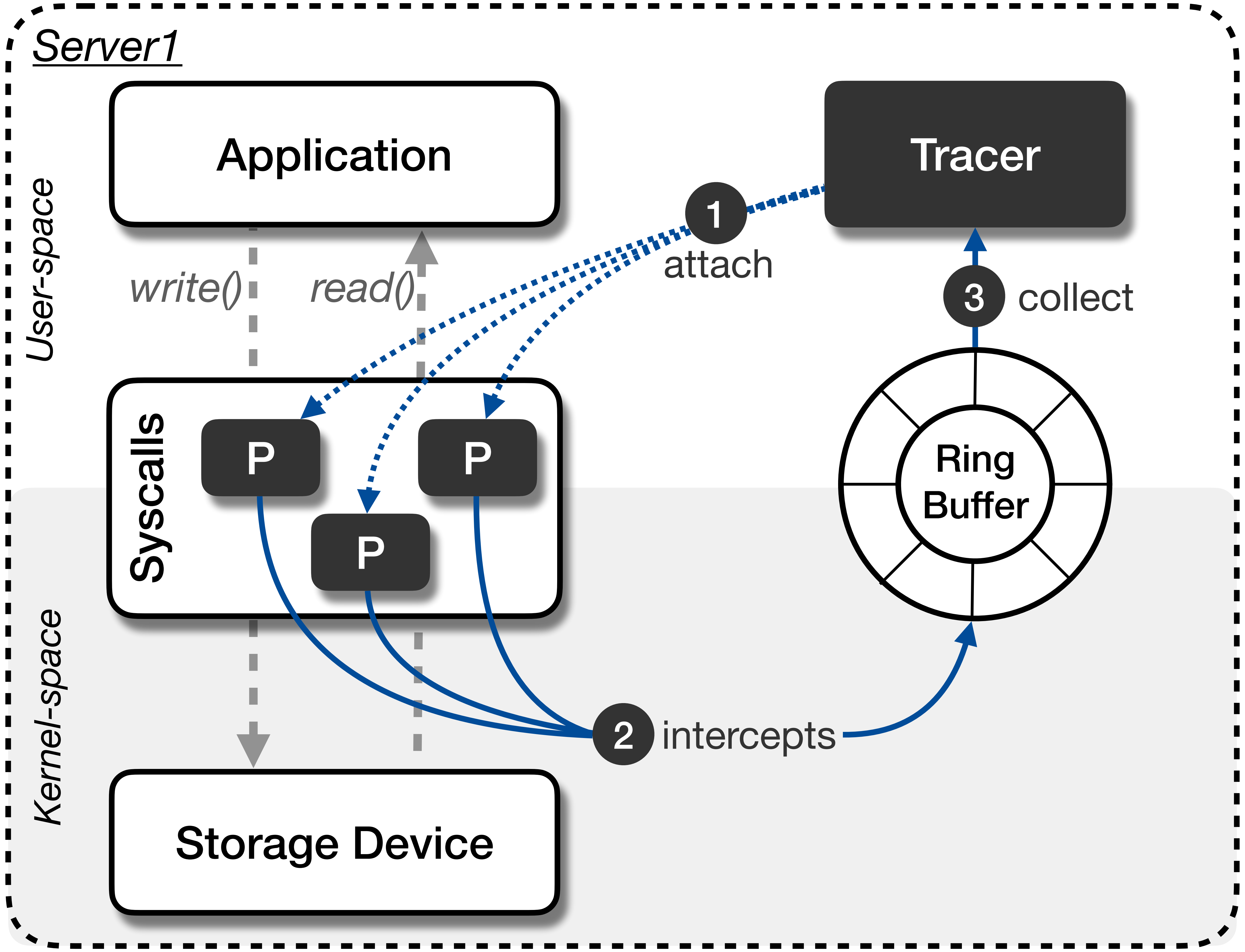
■ DIO's components ⟶ DIO main flow - - -▸ App flow

DIO Design



DIO's tracer runs along the targeted application, intercepting its syscalls

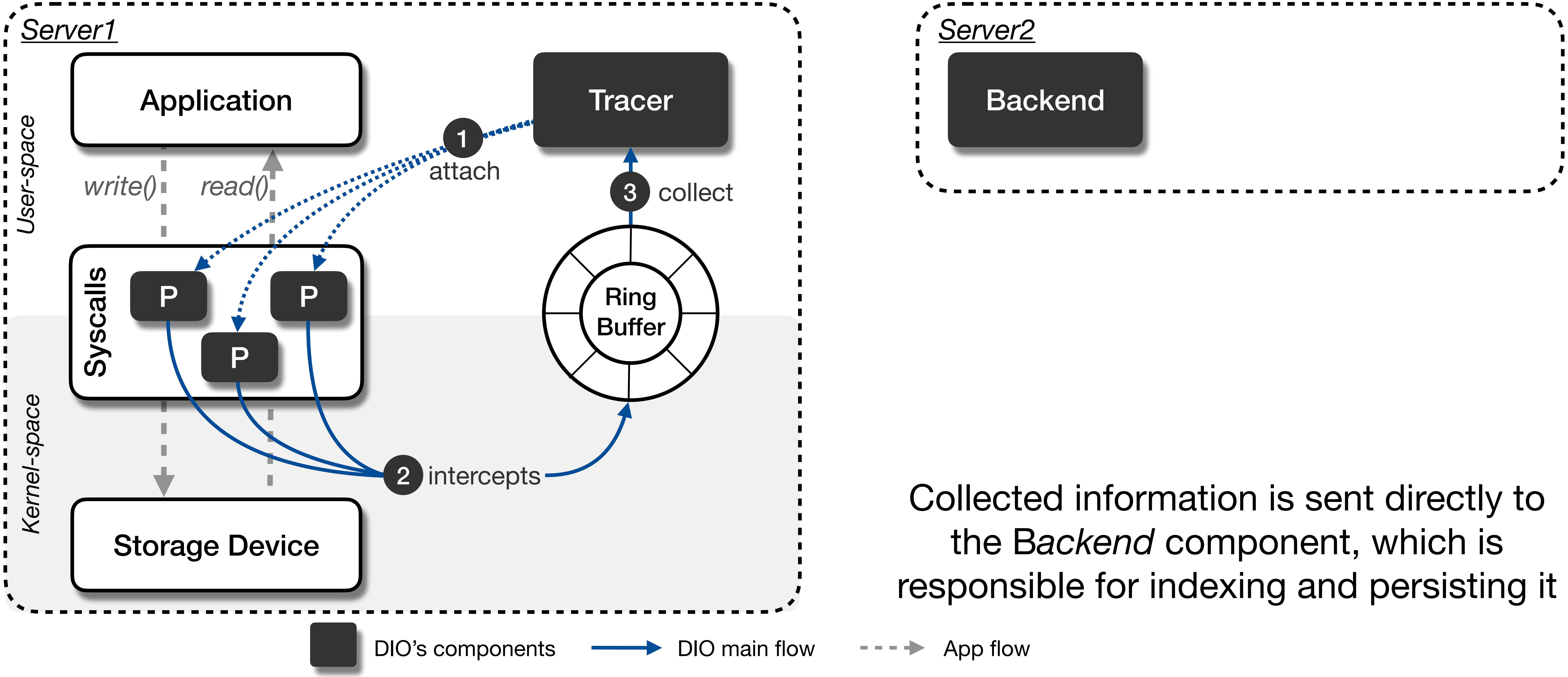
DIO Design



DIO's tracer runs along the targeted application, intercepting its syscalls

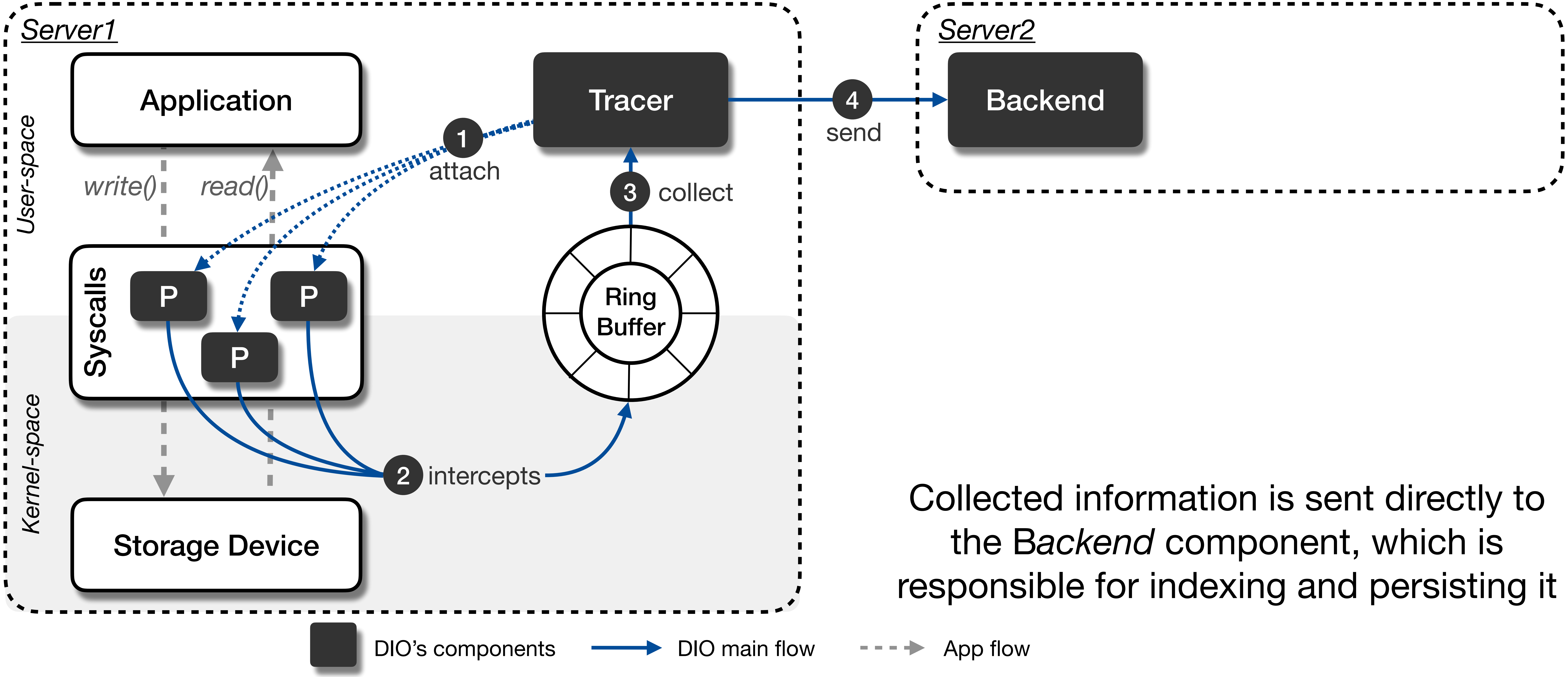
⬛ DIO's components ⟶ DIO main flow - - -▸ App flow

DIO Design



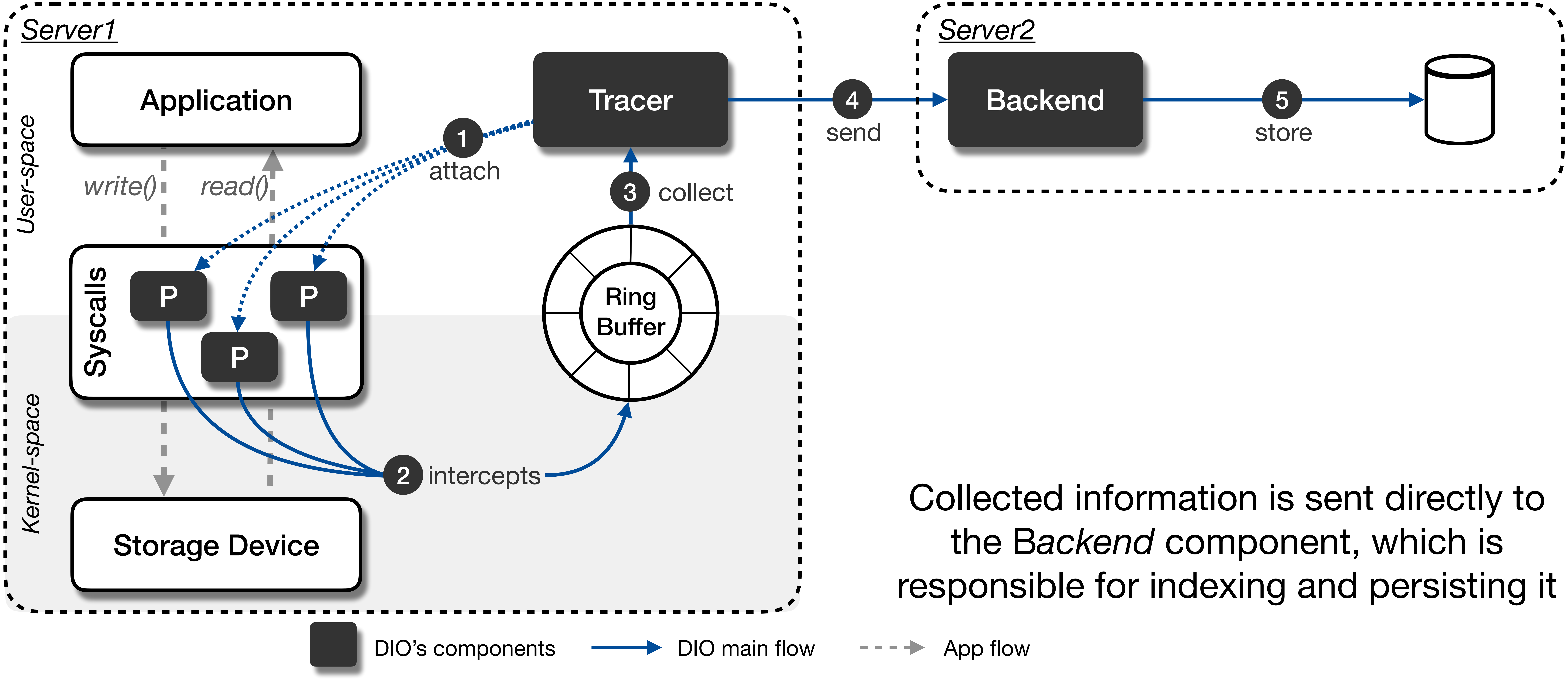
Collected information is sent directly to the *Backend* component, which is responsible for indexing and persisting it

DIO Design



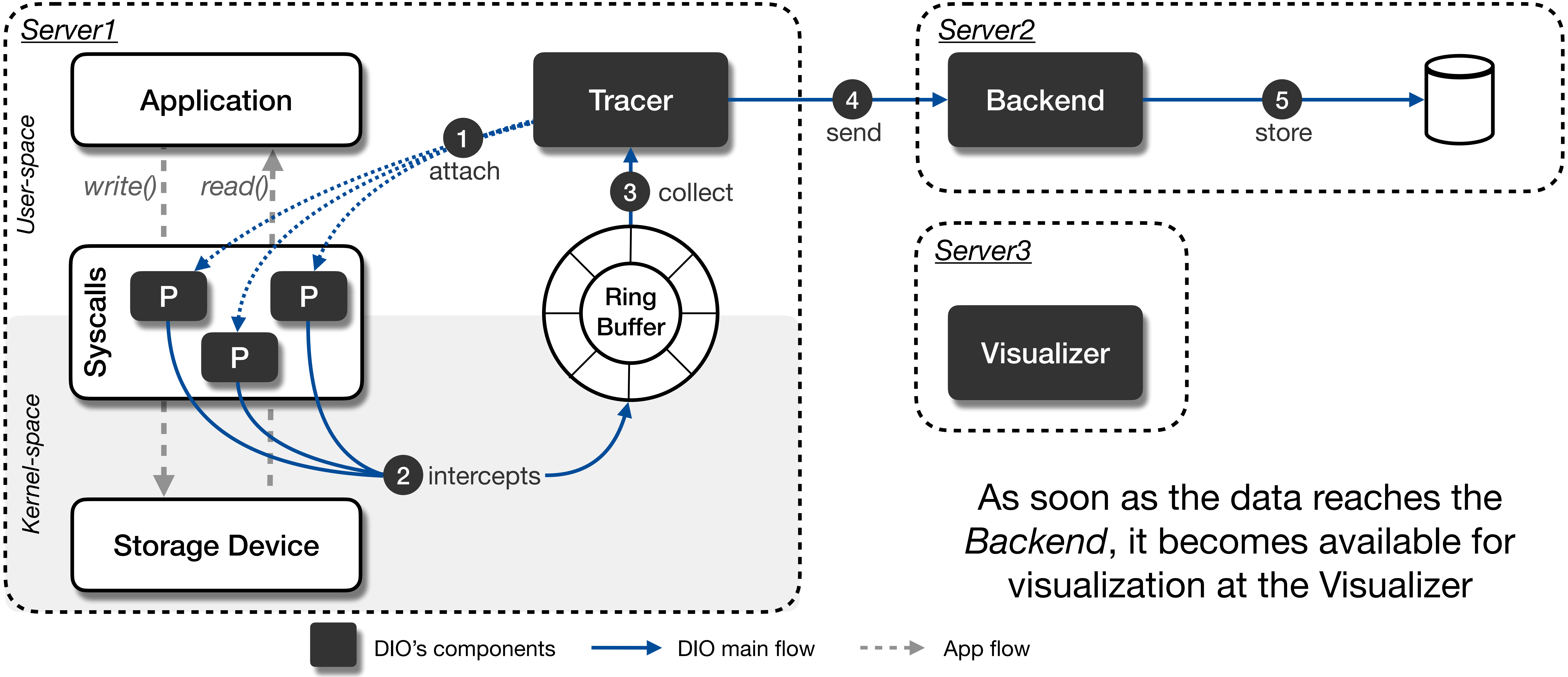
Collected information is sent directly to the *Backend* component, which is responsible for indexing and persisting it

DIO Design



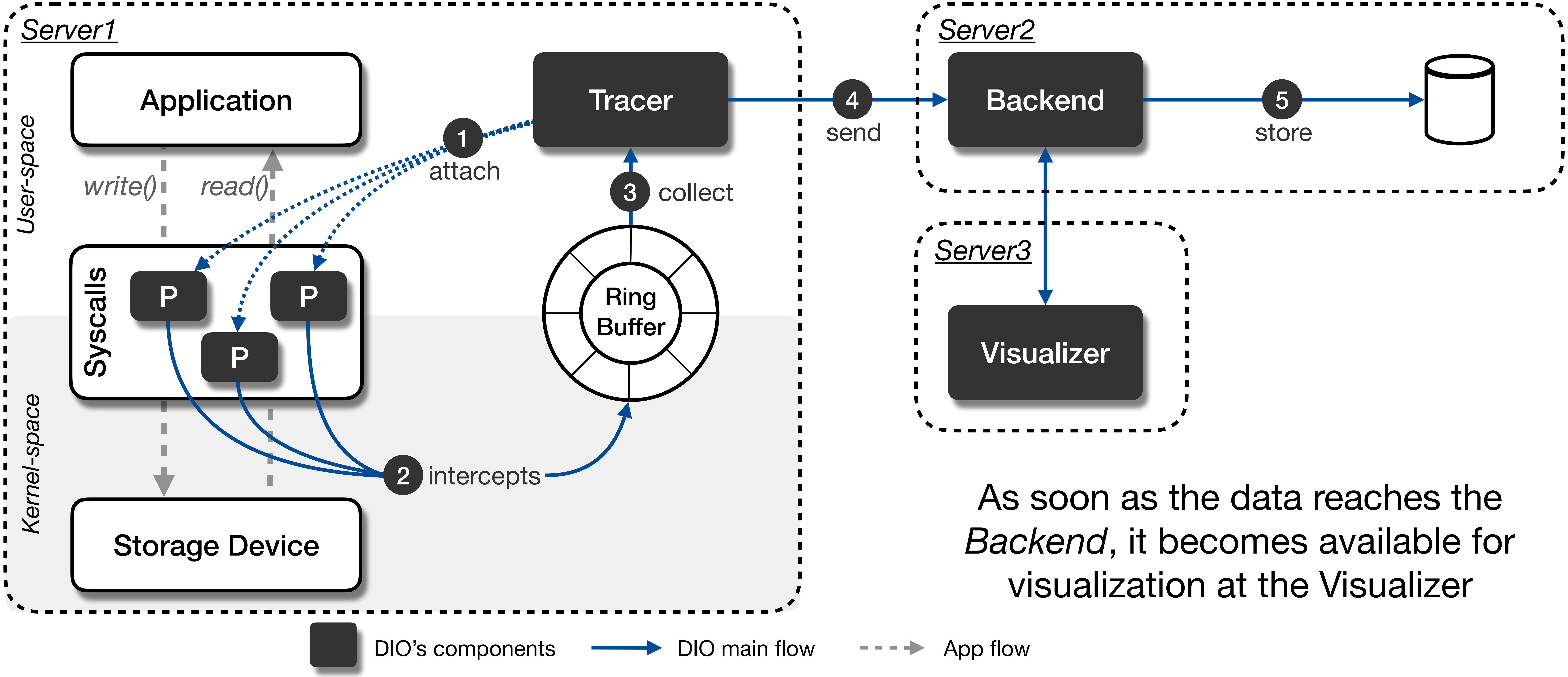
Collected information is sent directly to the *Backend* component, which is responsible for indexing and persisting it

DIO Design



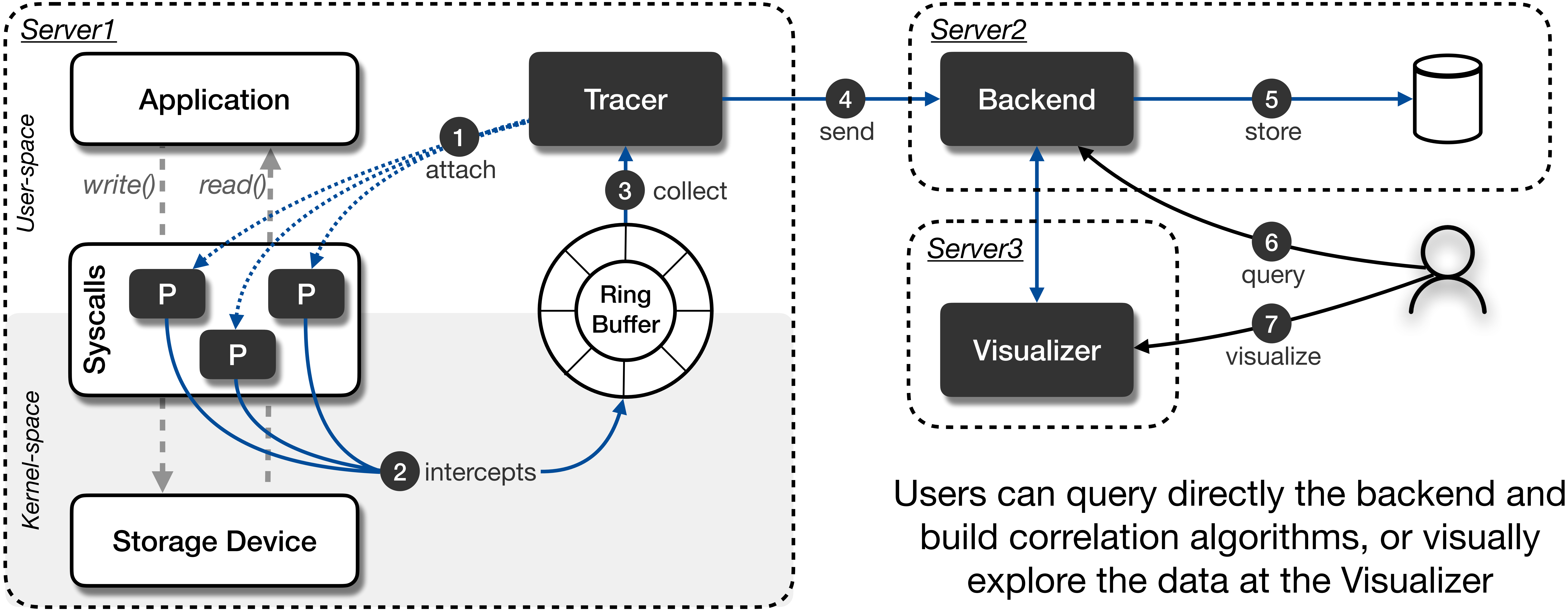
As soon as the data reaches the *Backend*, it becomes available for visualization at the Visualizer

DIO Design



As soon as the data reaches the *Backend*, it becomes available for visualization at the Visualizer

DIO Design



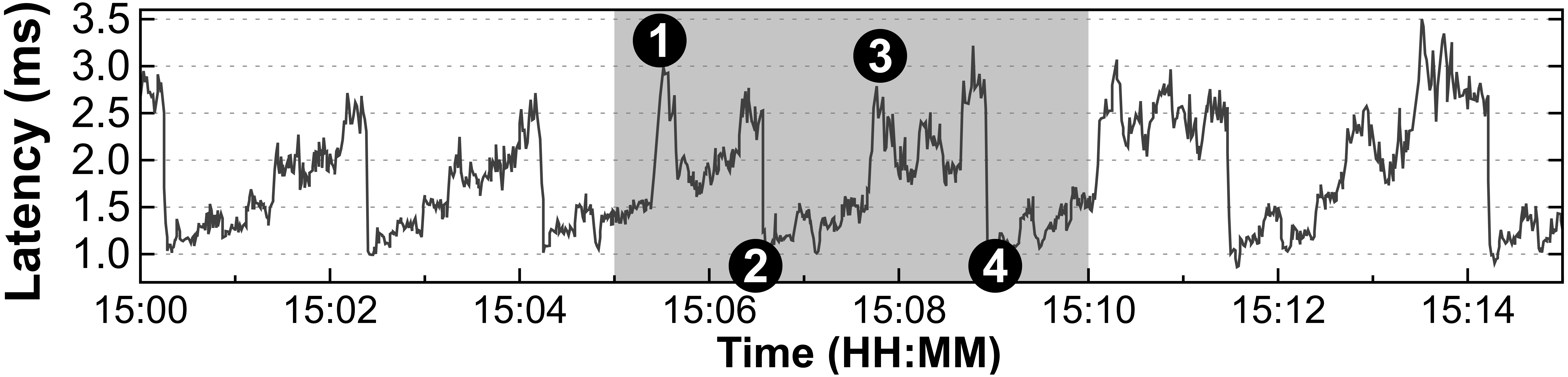
Users can query directly the backend and build correlation algorithms, or visually explore the data at the Visualizer

■ DIO's components —▶ DIO main flow - - -▶ App flow

DIO in Action

Finding the root cause of RocksDB’s performance anomalies

- **RocksDB:** An embedded key-value store
- **Problem:** RocksDB clients observe high tail latency (1 & 3)
  - Reproducible with db_bench benchmark

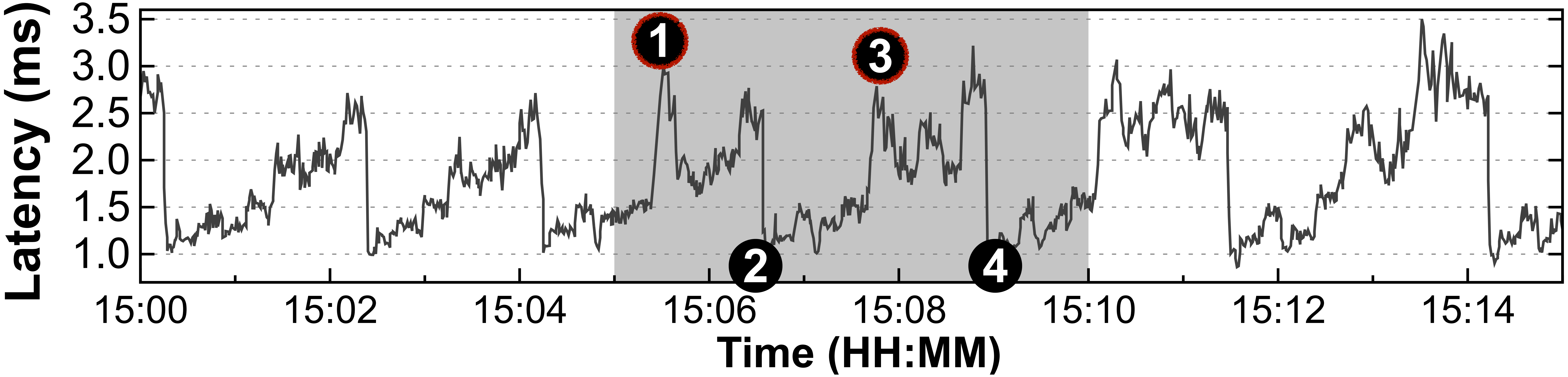


99th percentile latency for RocksDB client operations.

DIO in Action

Finding the root cause of RocksDB’s performance anomalies

- **RocksDB:** An embedded key-value store
- **Problem:** RocksDB clients observe high tail latency (1 & 3)
  - Reproducible with db_bench benchmark

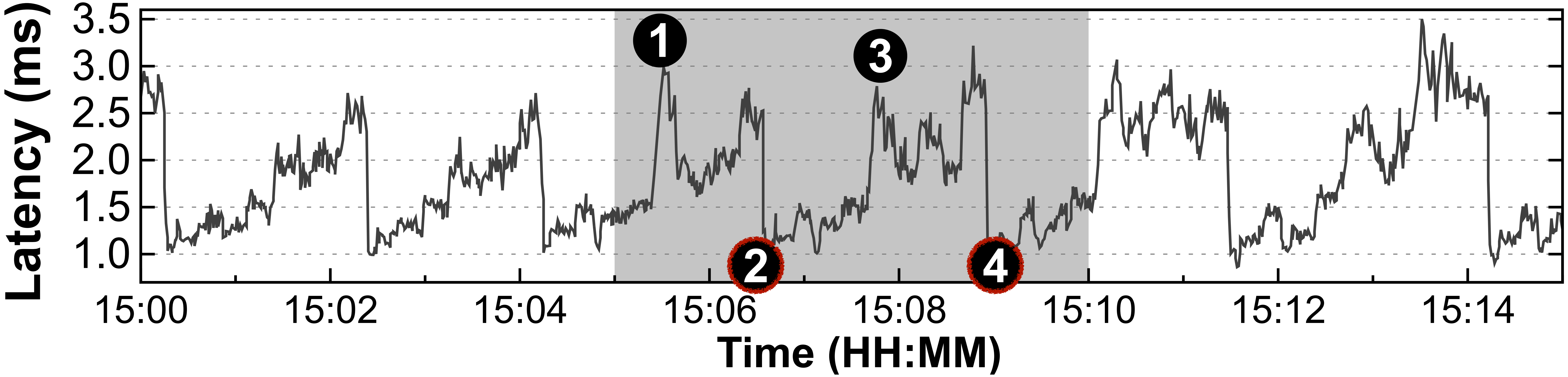


99th percentile latency for RocksDB client operations.

DIO in Action

Finding the root cause of RocksDB's performance anomalies

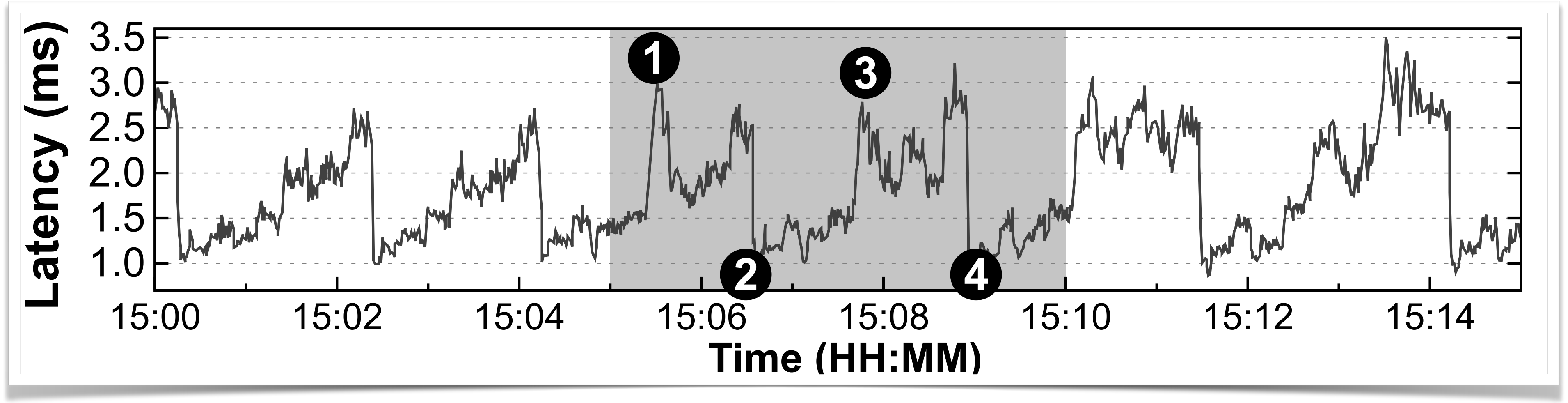
- **RocksDB:** An embedded key-value store
- **Problem:** RocksDB clients observe high tail latency (1 & 3)
  - Reproducible with db_bench benchmark

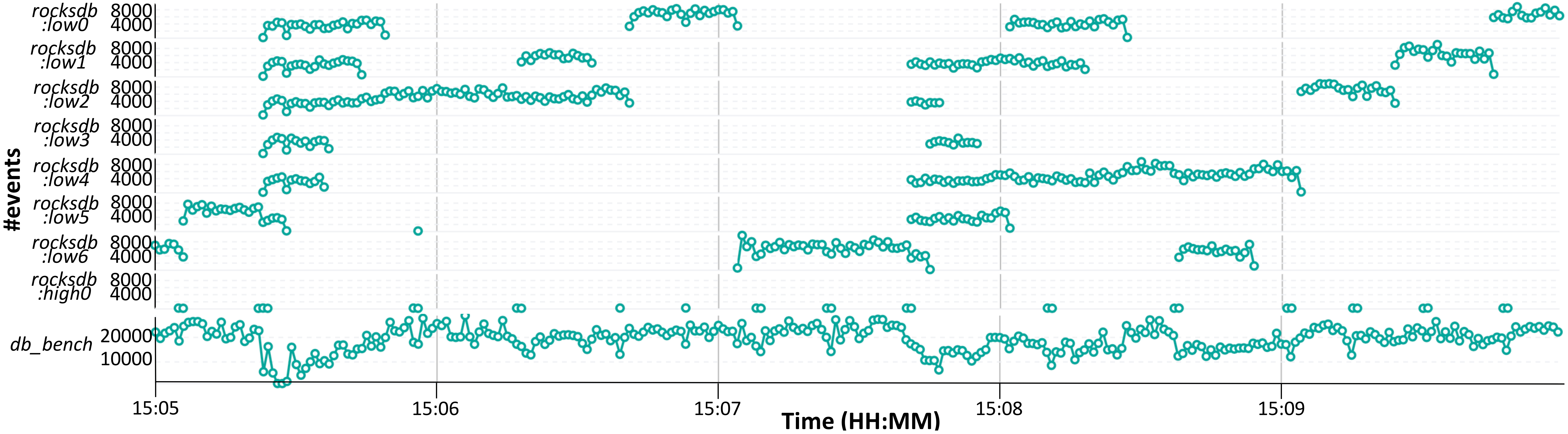


99th percentile latency for RocksDB client operations.

DIO in Action

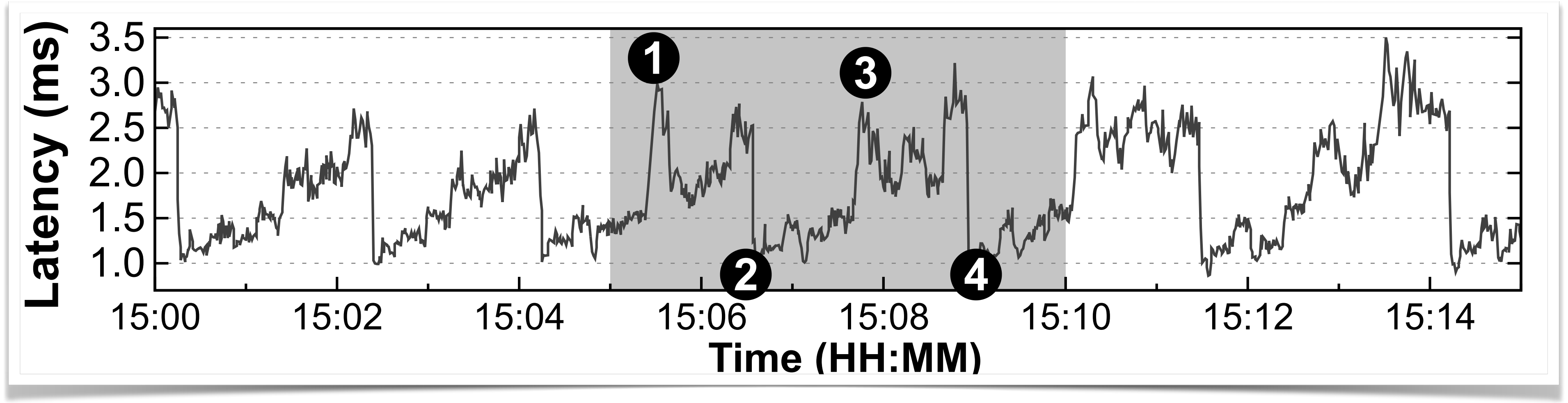
Finding the root cause of RocksDB’s performance anomalies



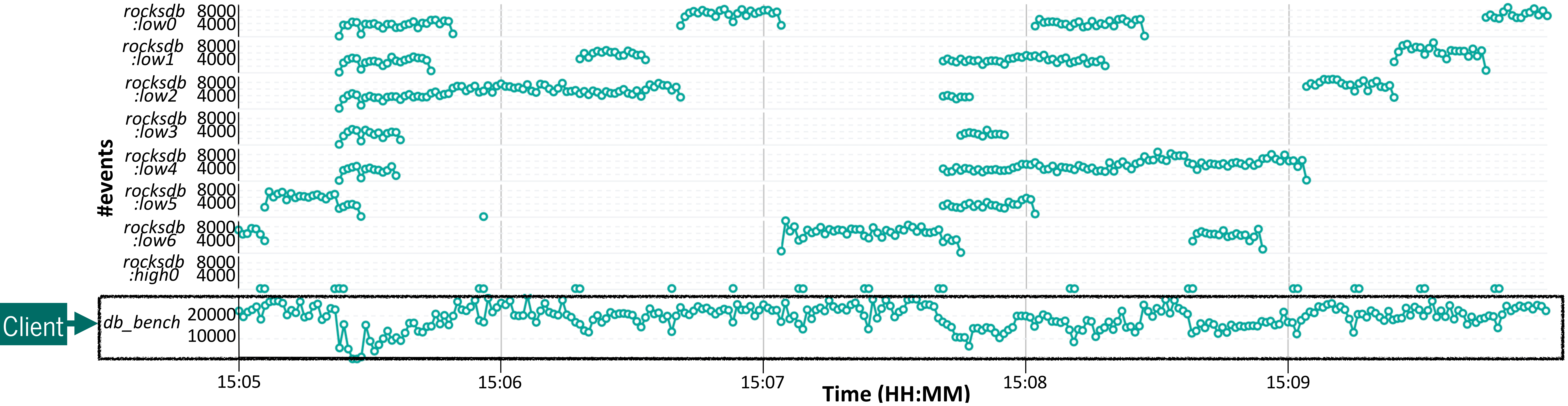


Syscalls issued by RocksDB over time, aggregated by thread name.

DIO in Action



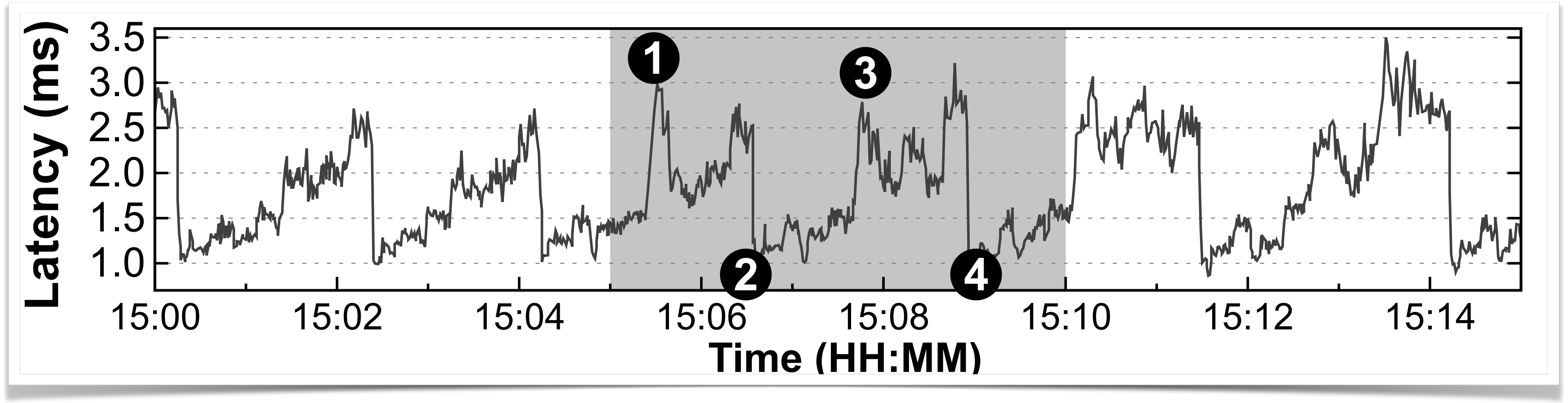
Finding the root cause of RocksDB’s performance anomalies

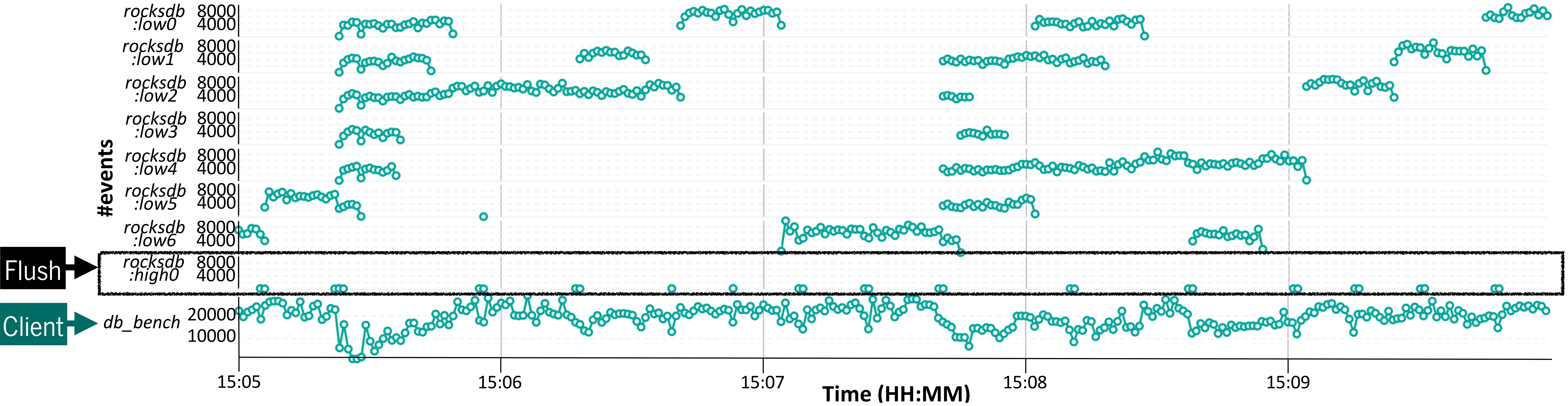


Syscalls issued by RocksDB over time, aggregated by thread name.

DIO in Action

Finding the root cause of RocksDB's performance anomalies

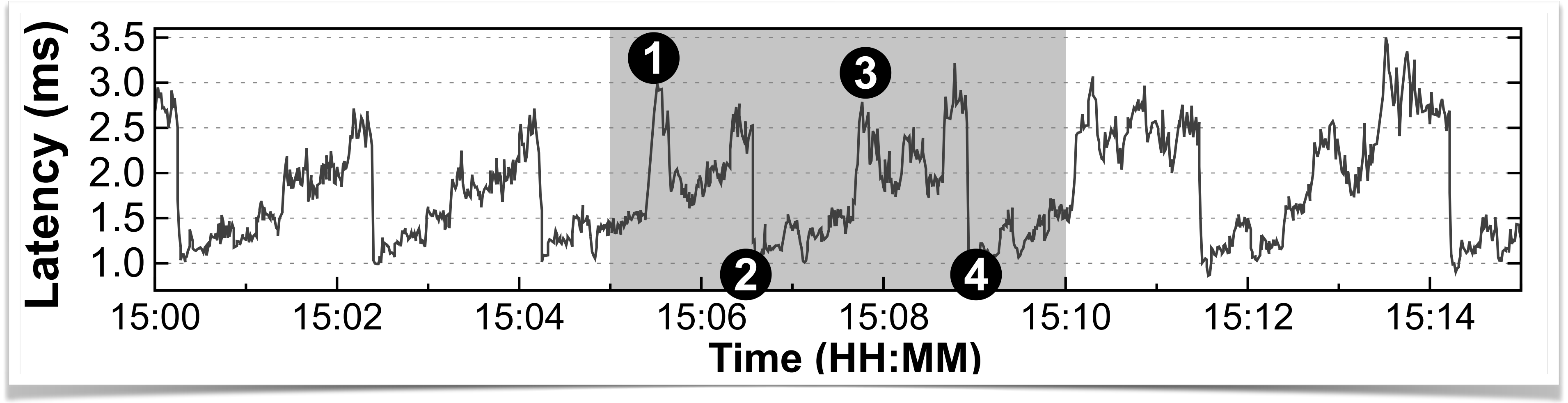


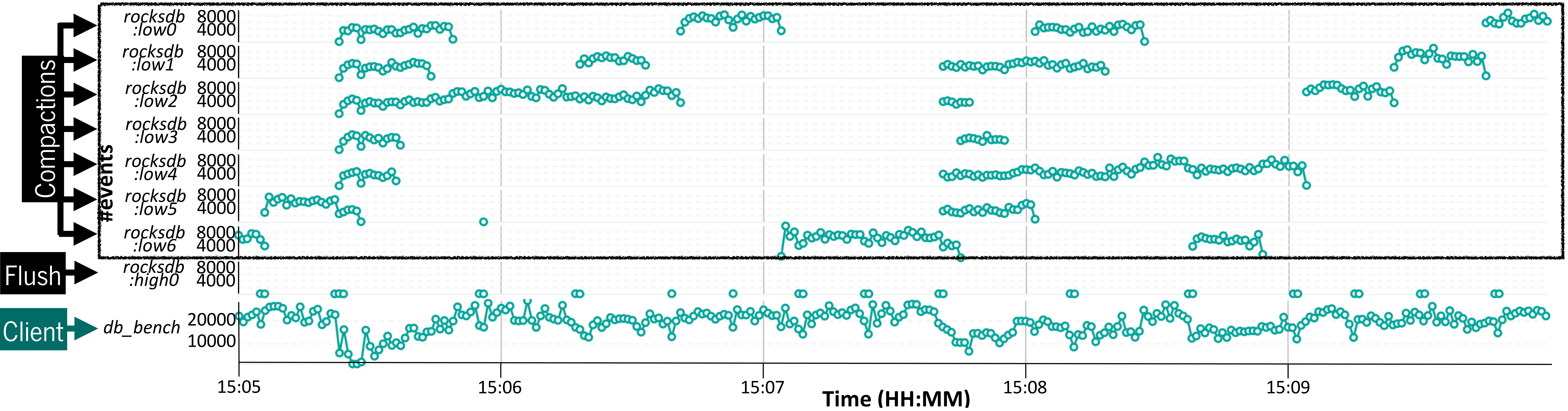


Syscalls issued by RocksDB over time, aggregated by thread name.

DIO in Action

Finding the root cause of RocksDB’s performance anomalies

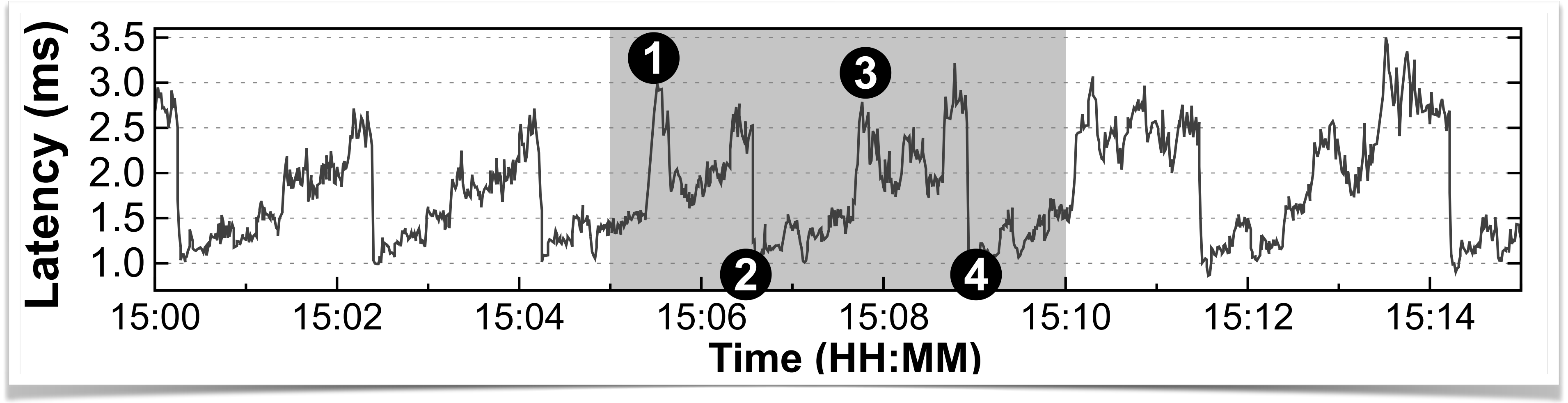


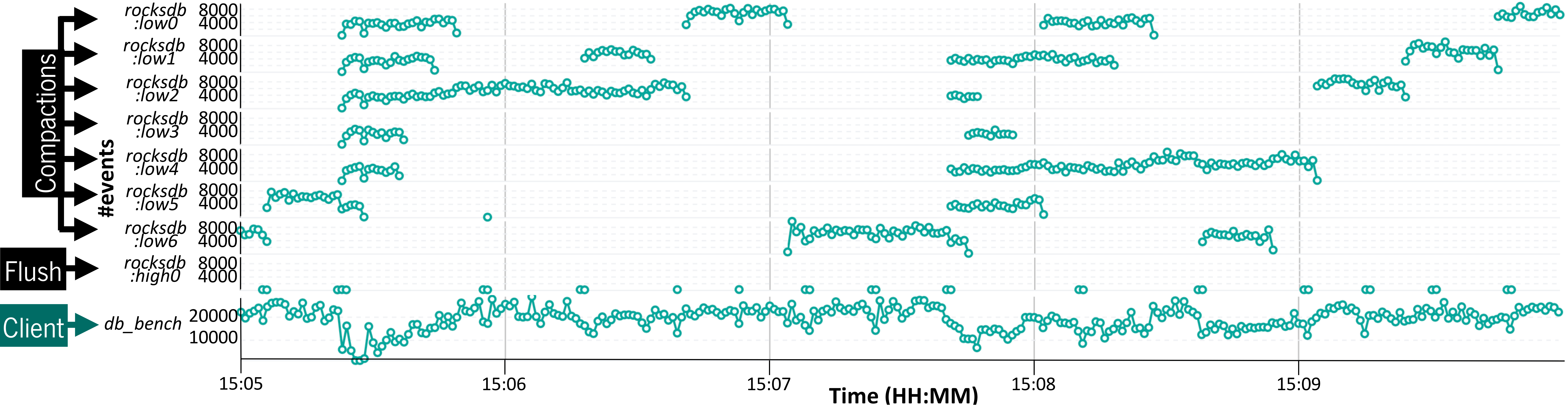


Syscalls issued by RocksDB over time, aggregated by thread name.

DIO in Action

Finding the root cause of RocksDB's performance anomalies

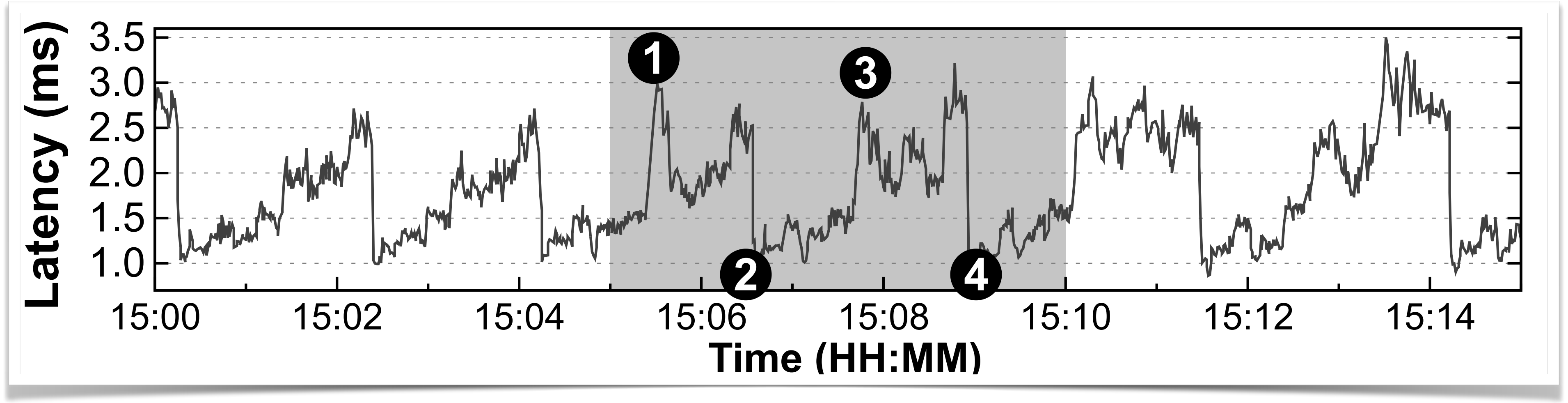


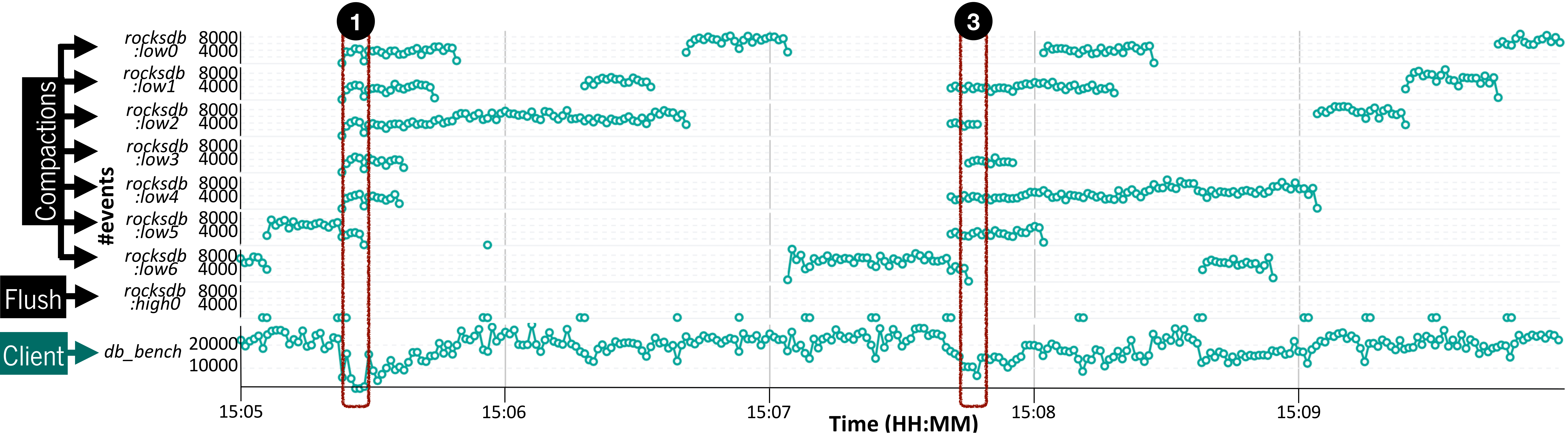


Syscalls issued by RocksDB over time, aggregated by thread name.

DIO in Action

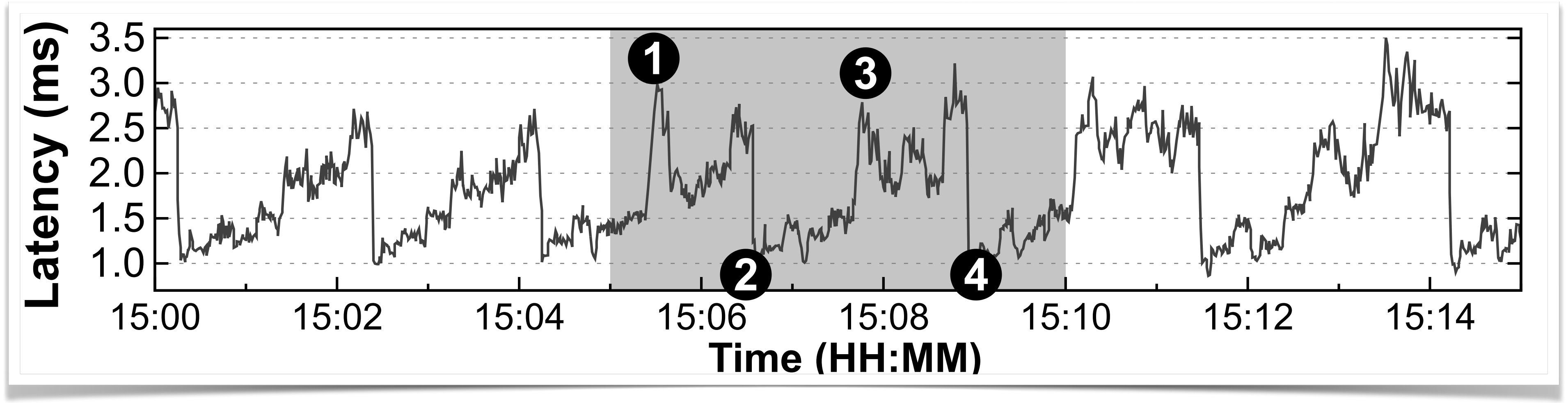
Finding the root cause of RocksDB’s performance anomalies





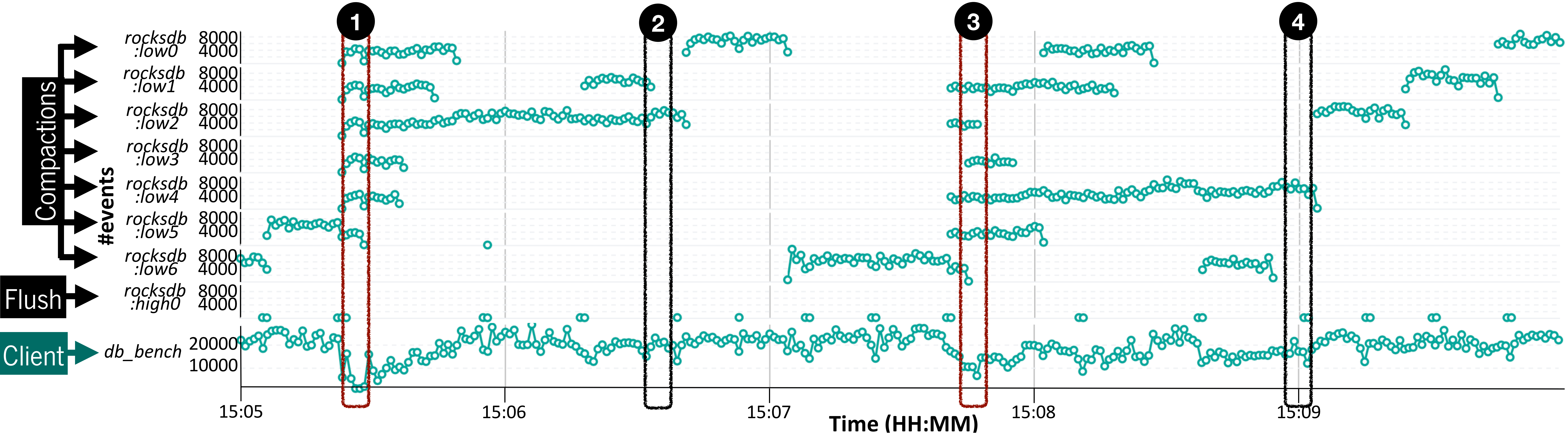
Syscalls issued by RocksDB over time, aggregated by thread name.

- (1&3) Multiple background threads perform I/O simultaneously, db_bench performance decreases



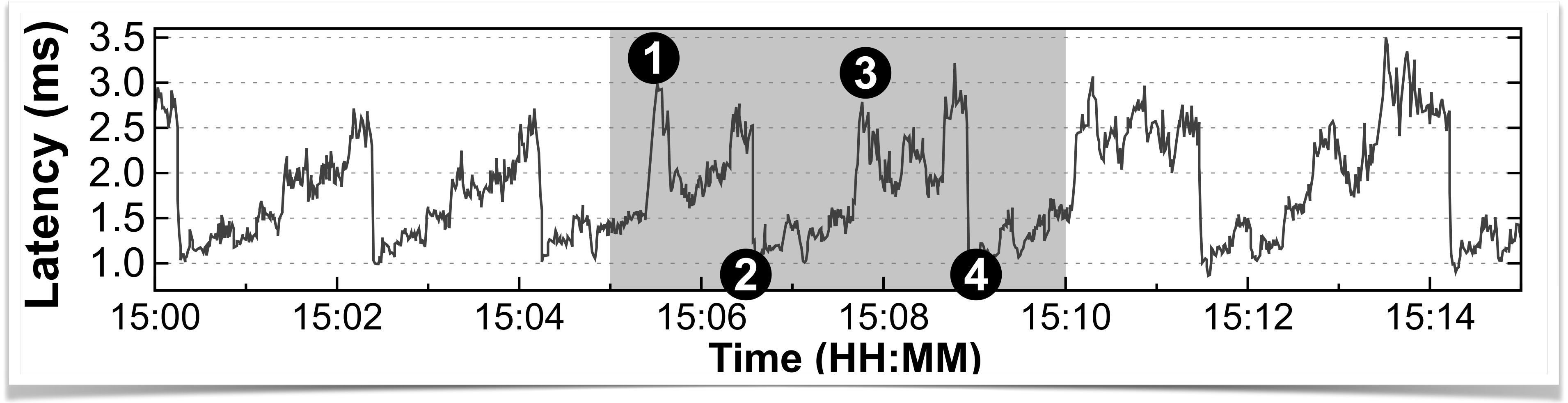
DIO in Action

Finding the root cause of RocksDB's performance anomalies



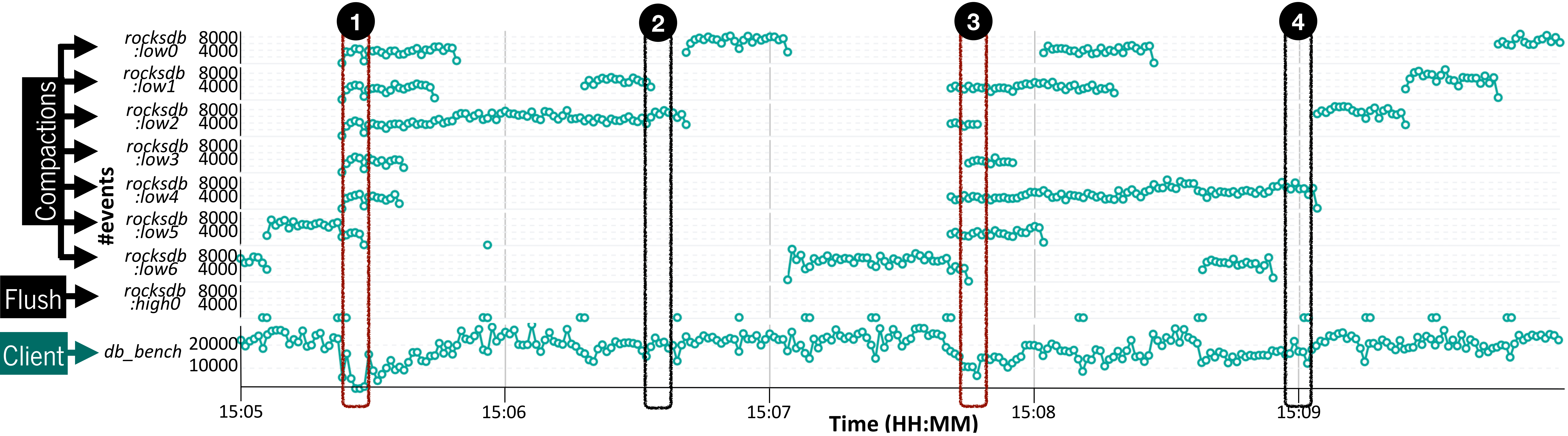
Syscalls issued by RocksDB over time, aggregated by thread name.

- (1&3) Multiple background threads perform I/O simultaneously, db_bench performance decreases
- (2&4) Few background threads perform I/O simultaneously, db_bench performance improves



DIO in Action

Finding the root cause of RocksDB's performance anomalies



Syscalls issued by RocksDB over time, aggregated by thread name.

Root Cause: Interference between client writes, flushes and compactions

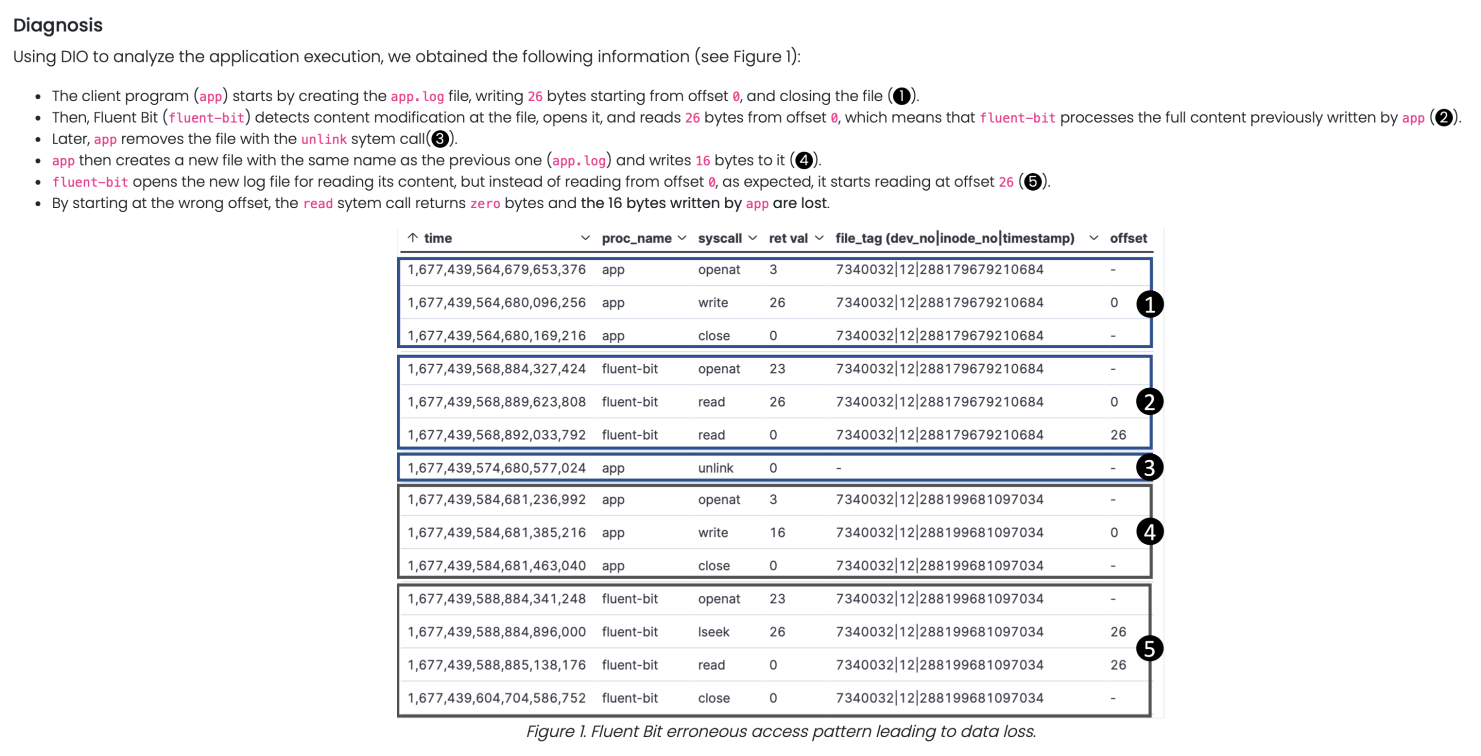
First observed in SILK (ATC'19) by instrumenting and manually inspecting more than 440K LoC

DIO Summary

- DIO enables the diagnosis of storage **correctness**, **dependability** and **performance** issues and avoids the need for combining multiple tools and running the application multiple times

Fluent Bit

Identification of erroneous actions that lead to data loss



Diagnosis

Using DIO to analyze the application execution, we obtained the following information (see Figure 1):

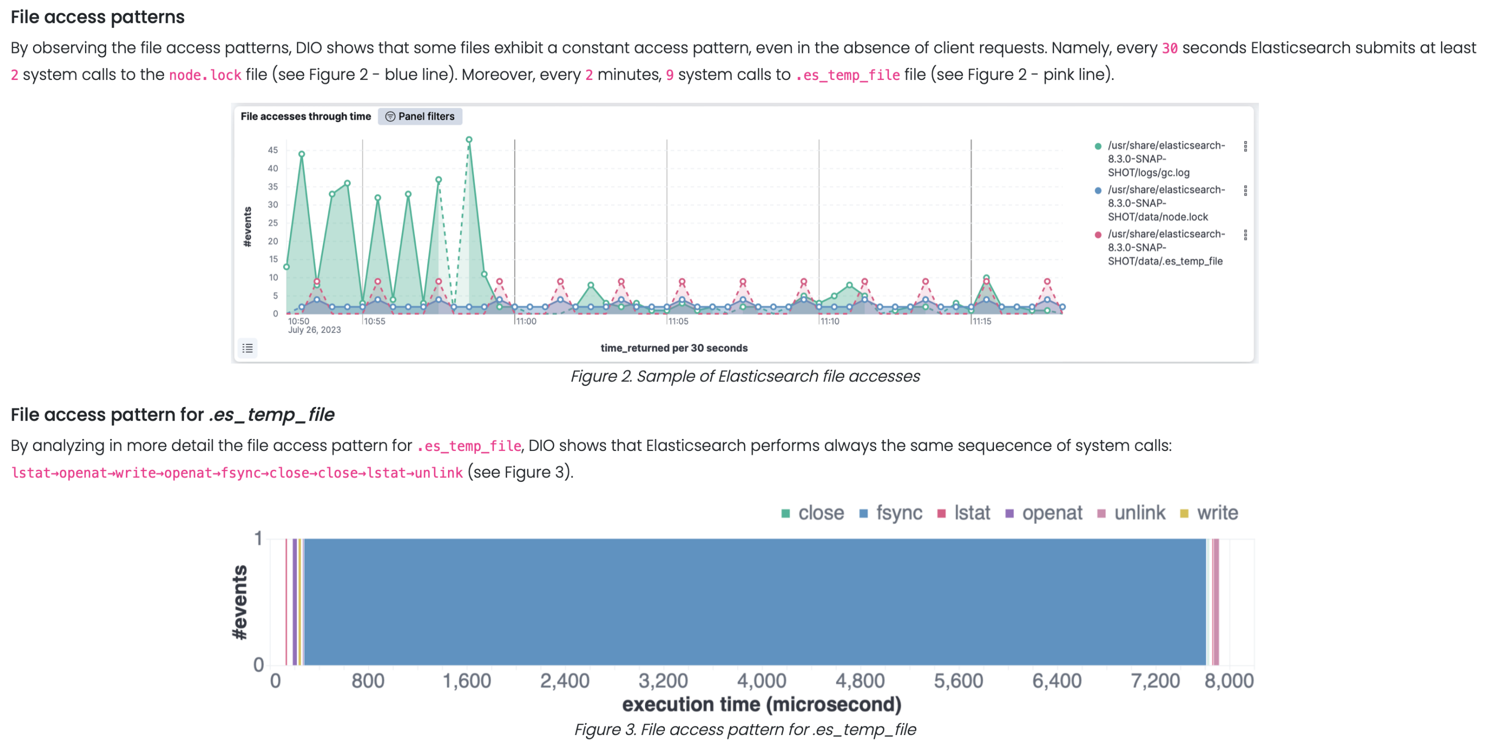
- The client program (app) starts by creating the app.log file, writing 26 bytes starting from offset 0, and closing the file (❶).
- Then, Fluent Bit (fluent-bit) detects content modification at the file, opens it, and reads 26 bytes from offset 0, which means that fluent-bit processes the full content previously written by app (❷).
- Later, app removes the file with the unlink sytem call(❸).
- app then creates a new file with the same name as the previous one (app.log) and writes 16 bytes to it (❹).
- fluent-bit opens the new log file for reading its content, but instead of reading from offset 0, as expected, it starts reading at offset 26 (❺).
- By starting at the wrong offset, the read sytem call returns zero bytes and **the 16 bytes written by app are lost.**

↑ time	proc_name	syscall	ret val	file_tag (dev_no|inode_no|timestamp)	offset	
1,677,439,564,679,653,376	app	openat	3	7340032|12|288179679210684	-	
1,677,439,564,680,096,256	app	write	26	7340032|12|288179679210684	0	❶
1,677,439,564,680,169,216	app	close	0	7340032|12|288179679210684	-	
1,677,439,568,884,327,424	fluent-bit	openat	23	7340032|12|288179679210684	-	
1,677,439,568,889,623,808	fluent-bit	read	26	7340032|12|288179679210684	0	❷
1,677,439,568,892,033,792	fluent-bit	read	0	7340032|12|288179679210684	26	
1,677,439,574,680,577,024	app	unlink	0	-	-	❸
1,677,439,584,681,236,992	app	openat	3	7340032|12|288199681097034	-	
1,677,439,584,681,385,216	app	write	16	7340032|12|288199681097034	0	❹
1,677,439,584,681,463,040	app	close	0	7340032|12|288199681097034	-	
1,677,439,588,884,341,248	fluent-bit	openat	23	7340032|12|288199681097034	-	
1,677,439,588,884,896,000	fluent-bit	lseek	26	7340032|12|288199681097034	26	❺
1,677,439,588,885,138,176	fluent-bit	read	0	7340032|12|288199681097034	26	
1,677,439,604,704,586,752	fluent-bit	close	0	7340032|12|288199681097034	-	

Figure 1. Fluent Bit erroneous access pattern leading to data loss.

Elasticsearch

Top-down exploration and I/O diagnosis



File access patterns

By observing the file access patterns, DIO shows that some files exhibit a constant access pattern, even in the absence of client requests. Namely, every 30 seconds Elasticsearch submits at least 2 system calls to the node.lock file (see Figure 2 - blue line). Moreover, every 2 minutes, 9 system calls to .es_temp_file file (see Figure 2 - pink line).

File accesses through time
Panel filters
#events
0
5
10
15
20
25
30
35
40
45
10:50
July 26, 2023
10:55
11:00
11:05
11:10
11:15
time_returned per 30 seconds
/usr/share/elasticsearch-8.3.0-SNAPSHOT/logs/gc.log
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/node.lock
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/.es_temp_file

Figure 2. Sample of Elasticsearch file accesses

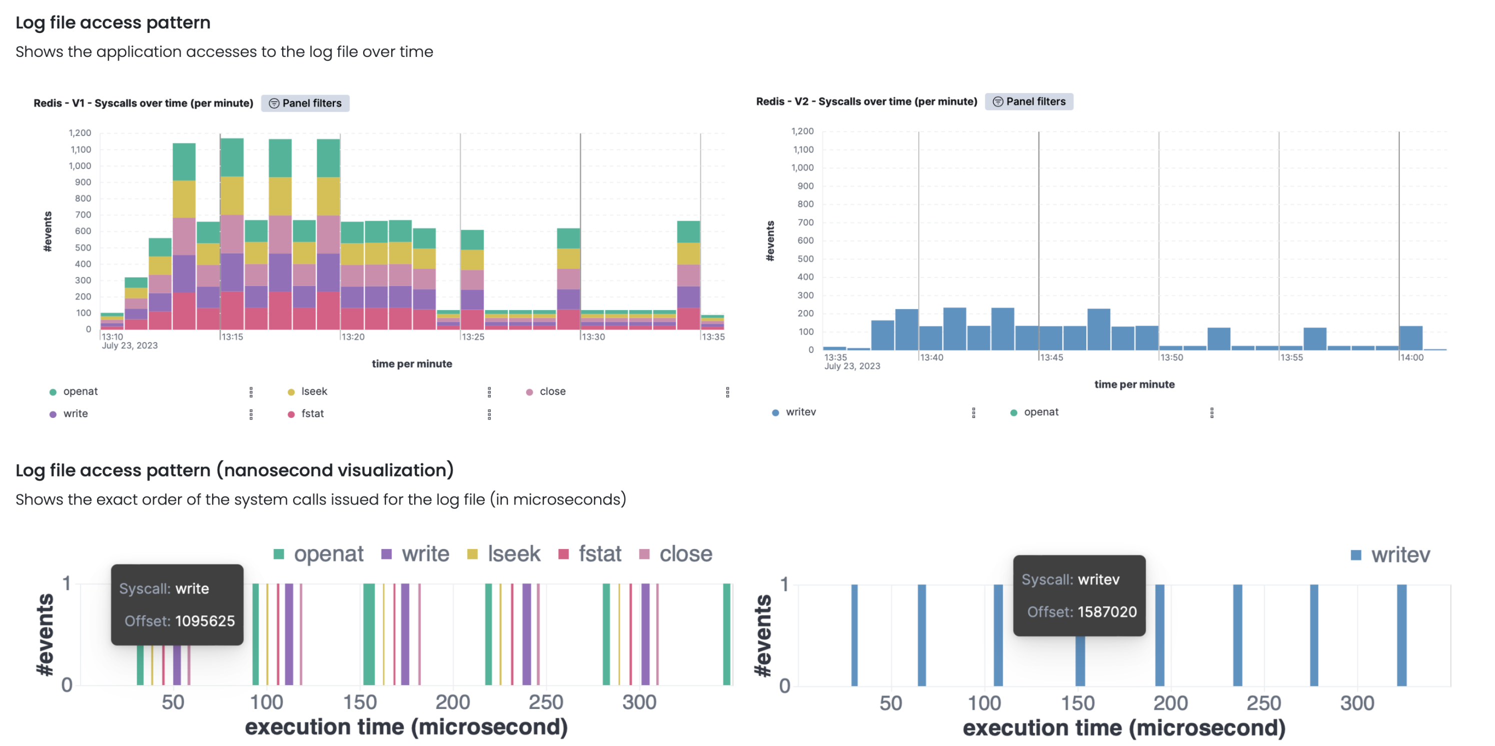
File access pattern for *.es_temp_file*

By analyzing in more detail the file access pattern for .es_temp_file, DIO shows that Elasticsearch performs always the same sequecence of system calls: lstat→openat→write→openat→fsync→close→close→lstat→unlink (see Figure 3).

Figure 3. File access pattern for .es_temp_file

Redis

Debugging and validation of inefficient I/O patterns



Log file access pattern

Shows the application accesses to the log file over time

Log file access pattern (nanosecond visualization)

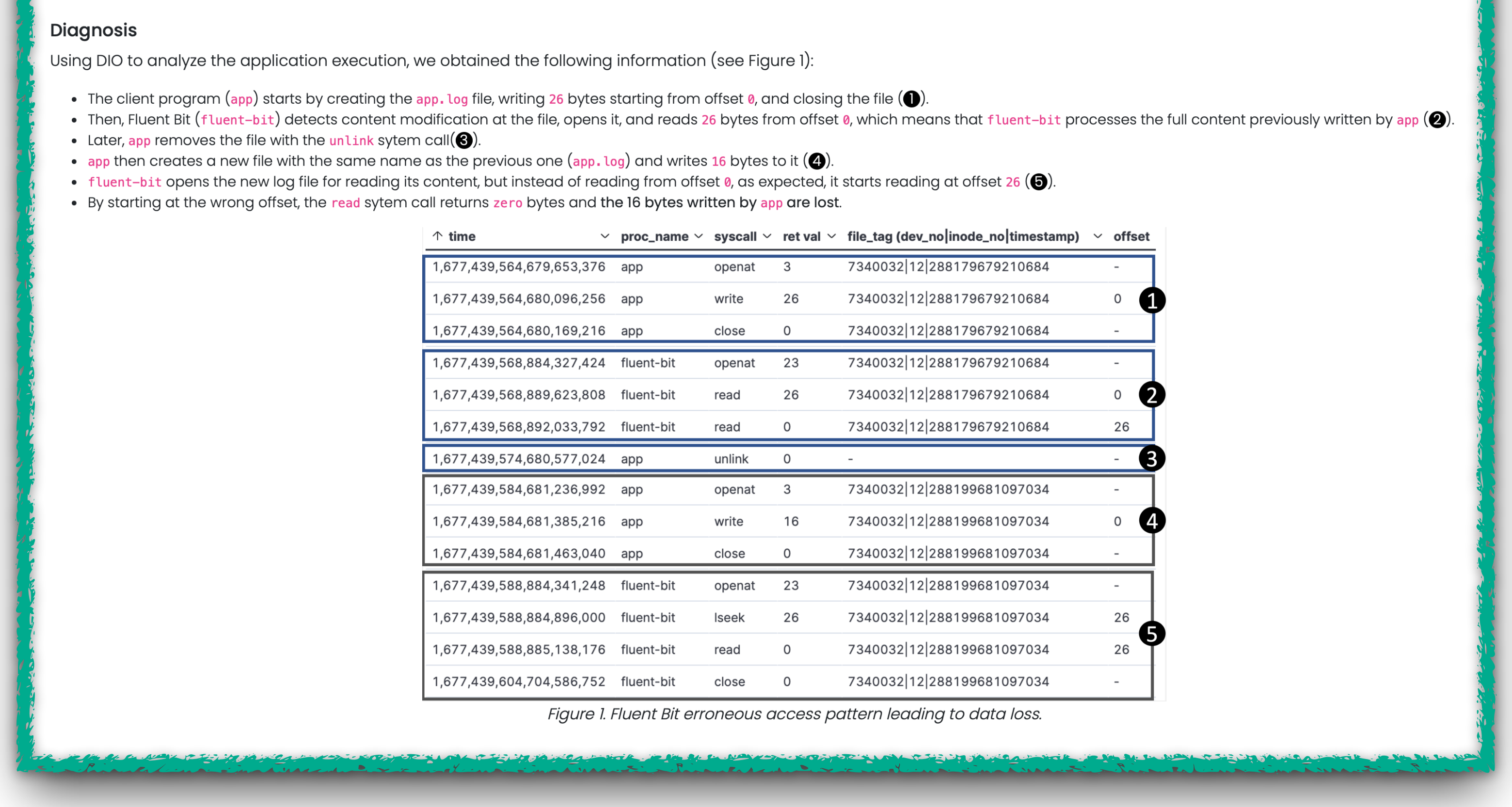
Shows the exact order of the system calls issued for the log file (in microseconds)

DIO Summary

- DIO enables the diagnosis of storage **correctness**, **dependability** and **performance** issues and avoids the need for combining multiple tools and running the application multiple times

Fluent Bit

Identification of erroneous actions that lead to data loss



Diagnosis

Using DIO to analyze the application execution, we obtained the following information (see Figure 1):

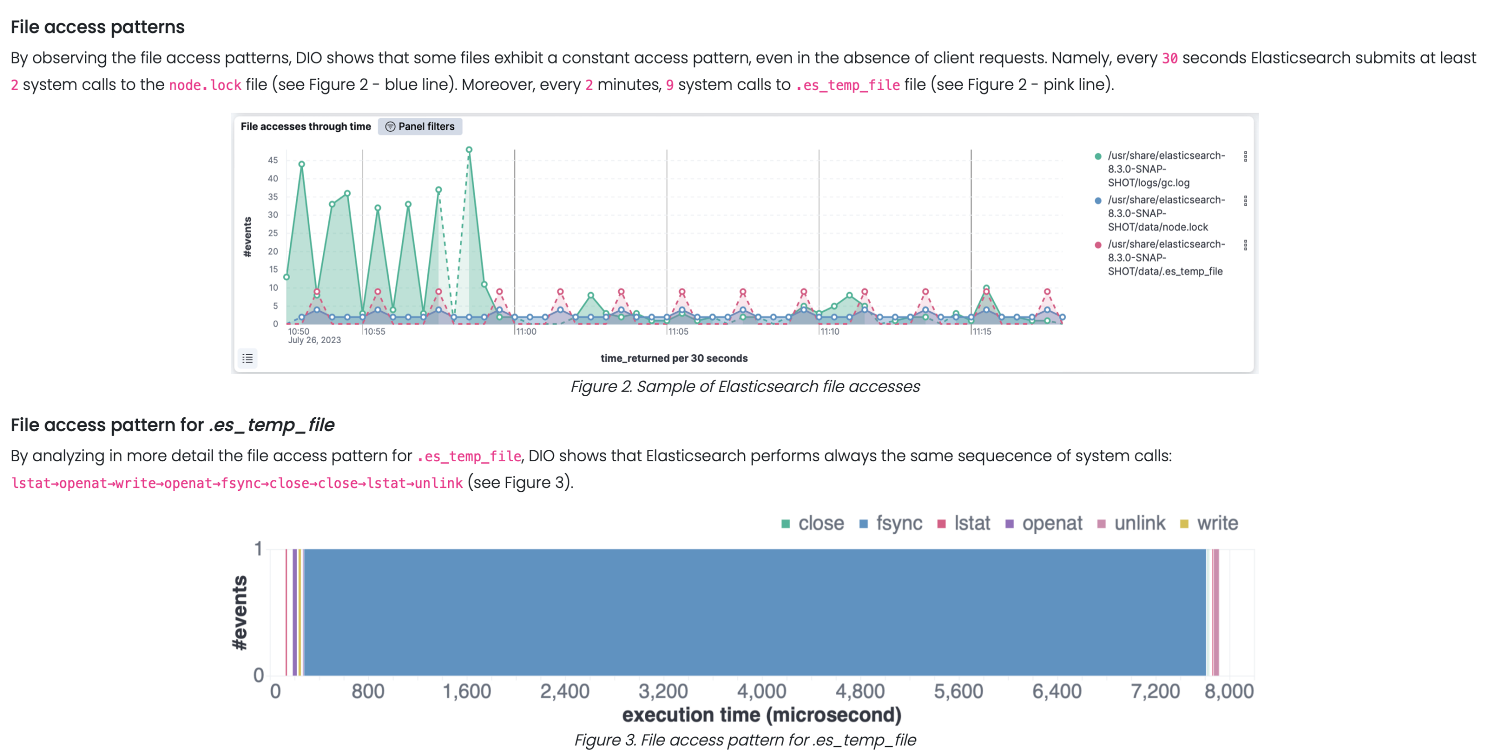
- The client program (app) starts by creating the app.log file, writing 26 bytes starting from offset 0, and closing the file (❶).
- Then, Fluent Bit (fluent-bit) detects content modification at the file, opens it, and reads 26 bytes from offset 0, which means that fluent-bit processes the full content previously written by app (❷).
- Later, app removes the file with the unlink sytem call(❸).
- app then creates a new file with the same name as the previous one (app.log) and writes 16 bytes to it (❹).
- fluent-bit opens the new log file for reading its content, but instead of reading from offset 0, as expected, it starts reading at offset 26 (❺).
- By starting at the wrong offset, the read sytem call returns zero bytes and **the 16 bytes written by app are lost.**

↑ time	proc_name	syscall	ret val	file_tag (dev_no|inode_no|timestamp)	offset	
1,677,439,564,679,653,376	app	openat	3	7340032|12|288179679210684	-	
1,677,439,564,680,096,256	app	write	26	7340032|12|288179679210684	0	❶
1,677,439,564,680,169,216	app	close	0	7340032|12|288179679210684	-	
1,677,439,568,884,327,424	fluent-bit	openat	23	7340032|12|288179679210684	-	
1,677,439,568,889,623,808	fluent-bit	read	26	7340032|12|288179679210684	0	❷
1,677,439,568,892,033,792	fluent-bit	read	0	7340032|12|288179679210684	26	
1,677,439,574,680,577,024	app	unlink	0	-	-	❸
1,677,439,584,681,236,992	app	openat	3	7340032|12|288199681097034	-	
1,677,439,584,681,385,216	app	write	16	7340032|12|288199681097034	0	❹
1,677,439,584,681,463,040	app	close	0	7340032|12|288199681097034	-	
1,677,439,588,884,341,248	fluent-bit	openat	23	7340032|12|288199681097034	-	
1,677,439,588,884,896,000	fluent-bit	lseek	26	7340032|12|288199681097034	26	❺
1,677,439,588,885,138,176	fluent-bit	read	0	7340032|12|288199681097034	26	
1,677,439,604,704,586,752	fluent-bit	close	0	7340032|12|288199681097034	-	

Figure 1. Fluent Bit erroneous access pattern leading to data loss.

Elasticsearch

Top-down exploration and I/O diagnosis



File access patterns

By observing the file access patterns, DIO shows that some files exhibit a constant access pattern, even in the absence of client requests. Namely, every 30 seconds Elasticsearch submits at least 2 system calls to the node.lock file (see Figure 2 - blue line). Moreover, every 2 minutes, 9 system calls to .es_temp_file file (see Figure 2 - pink line).

File accesses through time
Panel filters
#events
45
40
35
30
25
20
15
10
5
0
10:50
July 26, 2023
10:55
11:00
11:05
11:10
11:15
time_returned per 30 seconds
/usr/share/elasticsearch-8.3.0-SNAPSHOT/logs/gc.log
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/node.lock
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/.es_temp_file

Figure 2. Sample of Elasticsearch file accesses

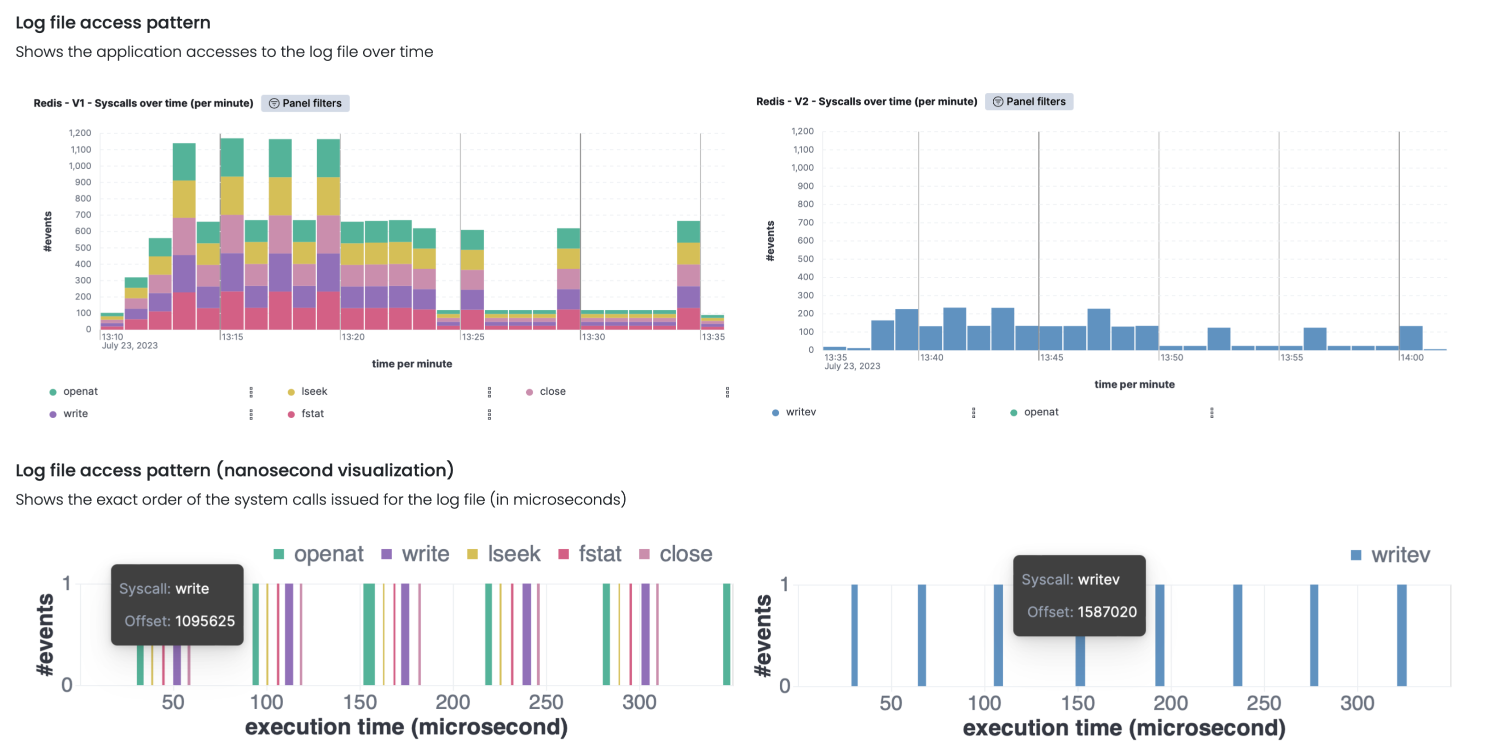
File access pattern for *.es_temp_file*

By analyzing in more detail the file access pattern for .es_temp_file, DIO shows that Elasticsearch performs always the same sequecence of system calls: lstat→openat→write→openat→fsync→close→close→lstat→unlink (see Figure 3).

Figure 3. File access pattern for .es_temp_file

Redis

Debugging and validation of inefficient I/O patterns



Log file access pattern

Shows the application accesses to the log file over time

Log file access pattern (nanosecond visualization)

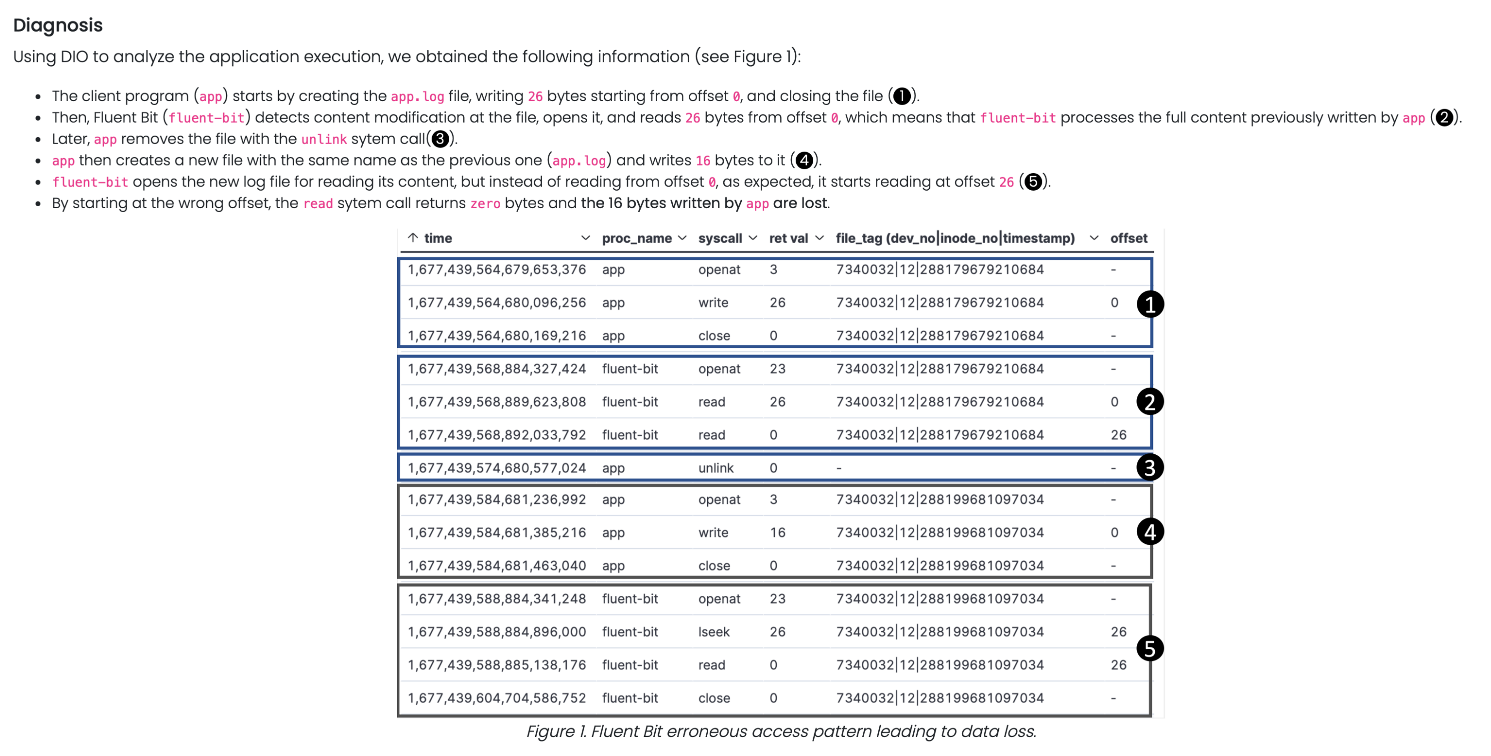
Shows the exact order of the system calls issued for the log file (in microseconds)

DIO Summary

- DIO enables the diagnosis of storage **correctness**, **dependability** and **performance** issues and avoids the need for combining multiple tools and running the application multiple times

Fluent Bit

Identification of erroneous actions that lead to data loss



Diagnosis

Using DIO to analyze the application execution, we obtained the following information (see Figure 1):

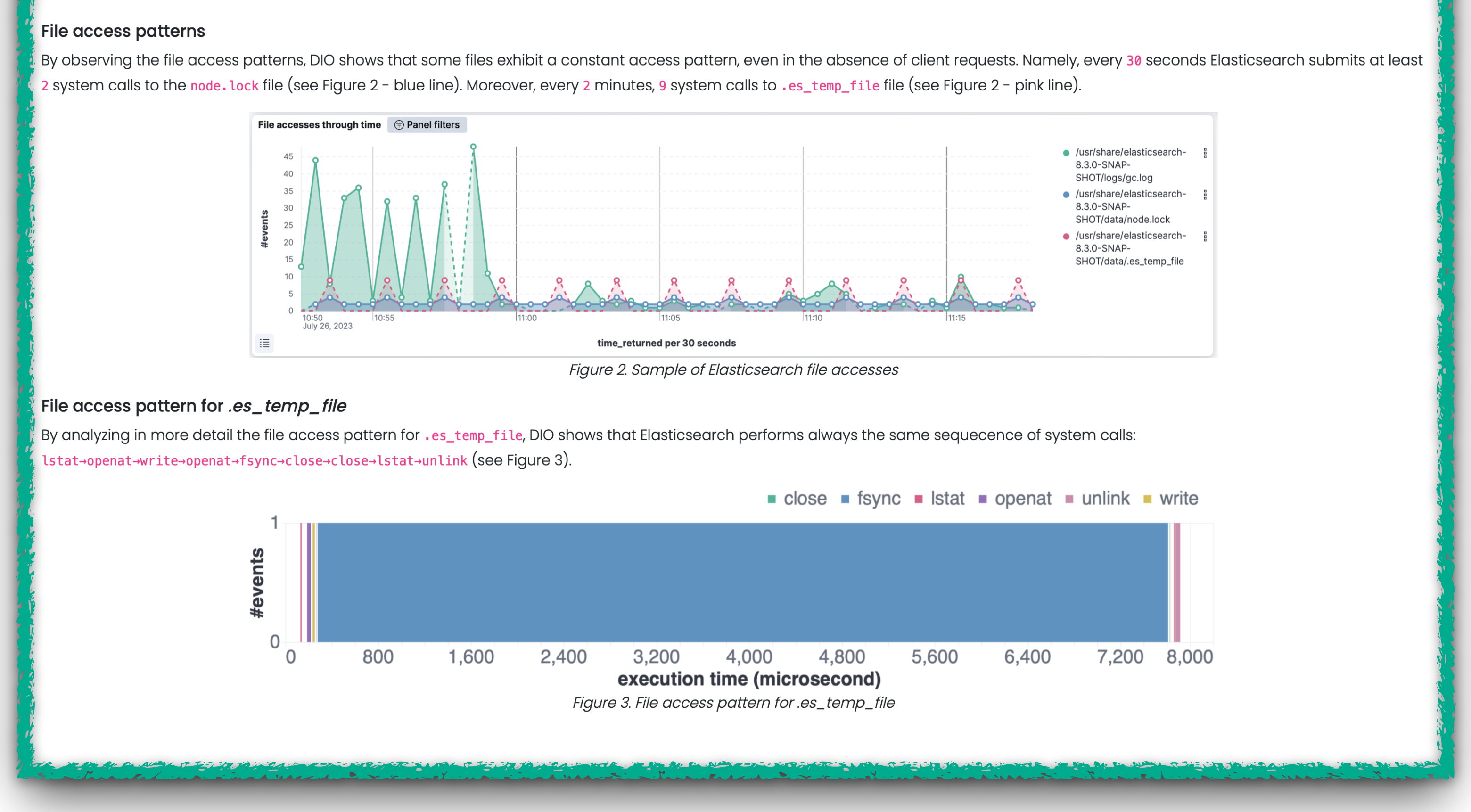
- The client program (app) starts by creating the app.log file, writing 26 bytes starting from offset 0, and closing the file (❶).
- Then, Fluent Bit (fluent-bit) detects content modification at the file, opens it, and reads 26 bytes from offset 0, which means that fluent-bit processes the full content previously written by app (❷).
- Later, app removes the file with the unlink sytem call(❸).
- app then creates a new file with the same name as the previous one (app.log) and writes 16 bytes to it (❹).
- fluent-bit opens the new log file for reading its content, but instead of reading from offset 0, as expected, it starts reading at offset 26 (❺).
- By starting at the wrong offset, the read sytem call returns zero bytes and **the 16 bytes written by app are lost.**

↑ time	proc_name	syscall	ret val	file_tag (dev_no|inode_no|timestamp)	offset	
1,677,439,564,679,653,376	app	openat	3	7340032|12|288179679210684	-	
1,677,439,564,680,096,256	app	write	26	7340032|12|288179679210684	0	❶
1,677,439,564,680,169,216	app	close	0	7340032|12|288179679210684	-	
1,677,439,568,884,327,424	fluent-bit	openat	23	7340032|12|288179679210684	-	
1,677,439,568,889,623,808	fluent-bit	read	26	7340032|12|288179679210684	0	❷
1,677,439,568,892,033,792	fluent-bit	read	0	7340032|12|288179679210684	26	
1,677,439,574,680,577,024	app	unlink	0	-	-	❸
1,677,439,584,681,236,992	app	openat	3	7340032|12|288199681097034	-	
1,677,439,584,681,385,216	app	write	16	7340032|12|288199681097034	0	❹
1,677,439,584,681,463,040	app	close	0	7340032|12|288199681097034	-	
1,677,439,588,884,341,248	fluent-bit	openat	23	7340032|12|288199681097034	-	
1,677,439,588,884,896,000	fluent-bit	lseek	26	7340032|12|288199681097034	26	
1,677,439,588,885,138,176	fluent-bit	read	0	7340032|12|288199681097034	26	❺
1,677,439,604,704,586,752	fluent-bit	close	0	7340032|12|288199681097034	-	

Figure 1. Fluent Bit erroneous access pattern leading to data loss.

Elasticsearch

Top-down exploration and I/O diagnosis



File access patterns

By observing the file access patterns, DIO shows that some files exhibit a constant access pattern, even in the absence of client requests. Namely, every 30 seconds Elasticsearch submits at least 2 system calls to the node.lock file (see Figure 2 - blue line). Moreover, every 2 minutes, 9 system calls to .es_temp_file file (see Figure 2 - pink line).

File accesses through time
Panel filters
#events
45
40
35
30
25
20
15
10
5
0
10:50
July 26, 2023
10:55
11:00
11:05
11:10
11:15
time_returned per 30 seconds
/usr/share/elasticsearch-8.3.0-SNAPSHOT/logs/gc.log
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/node.lock
/usr/share/elasticsearch-8.3.0-SNAPSHOT/data/.es_temp_file

Figure 2. Sample of Elasticsearch file accesses

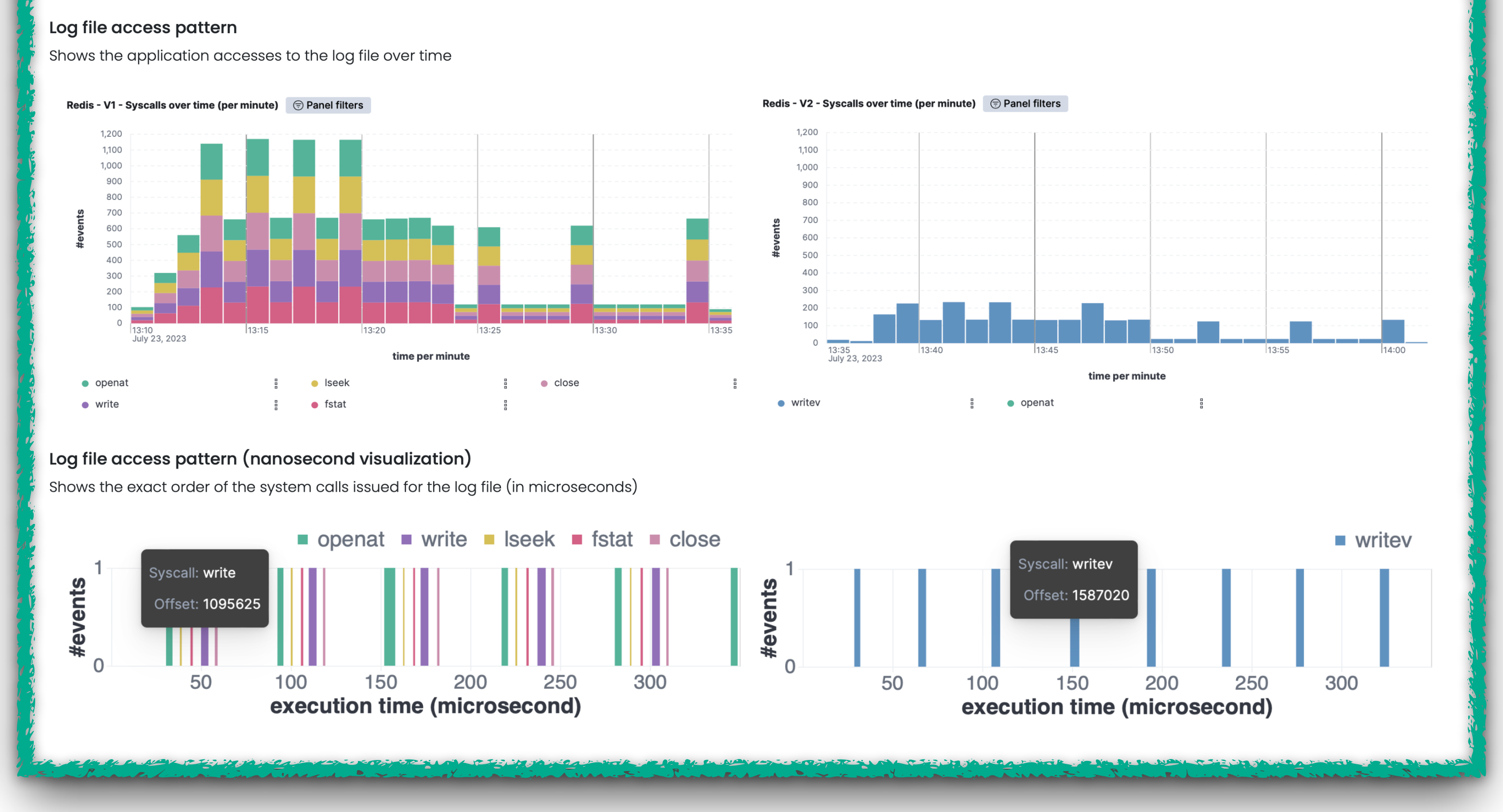
File access pattern for *.es_temp_file*

By analyzing in more detail the file access pattern for .es_temp_file, DIO shows that Elasticsearch performs always the same sequecence of system calls: lstat→openat→write→openat→fsync→close→close→lstat→unlink (see Figure 3).

Figure 3. File access pattern for .es_temp_file

Redis

Debugging and validation of inefficient I/O patterns



Log file access pattern

Shows the application accesses to the log file over time

Log file access pattern (nanosecond visualization)

Shows the exact order of the system calls issued for the log file (in microseconds)

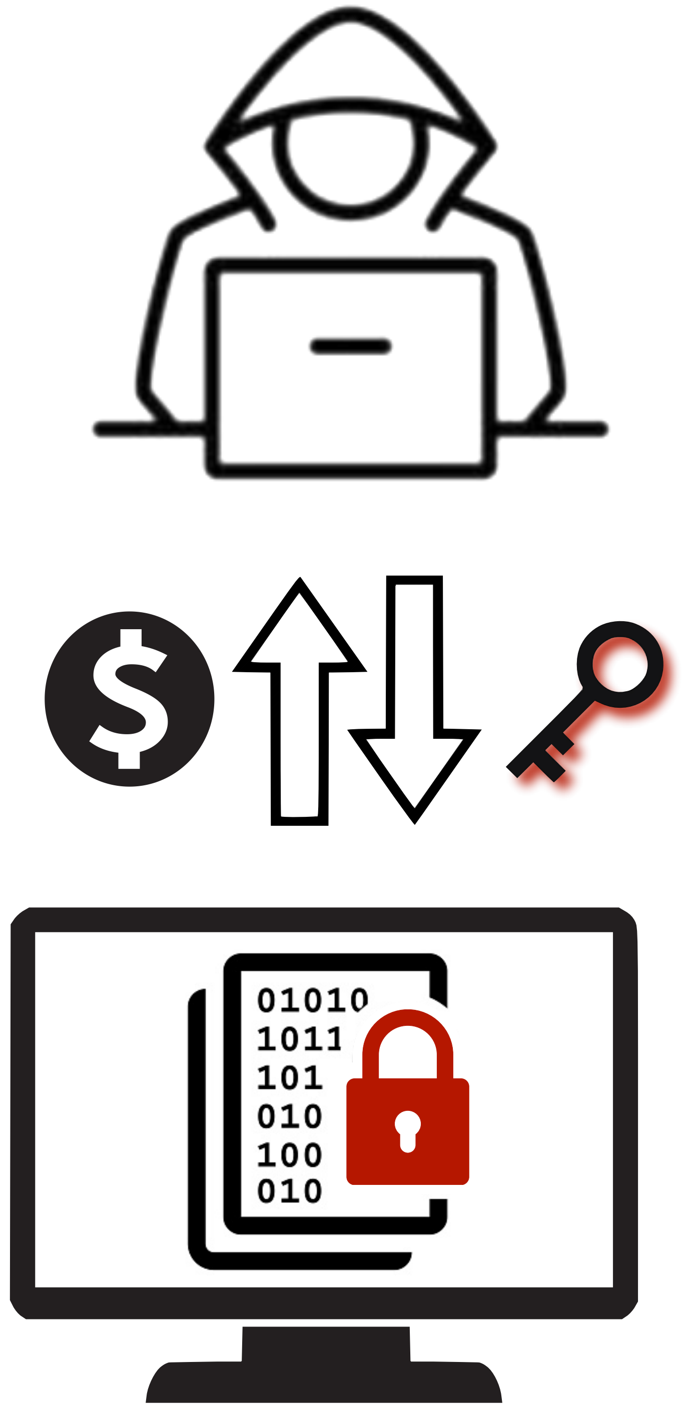
Custom and Improved Analysis

CRIBA, a tool for diagnosing the I/O behavior of Linux cryptographic ransomware

- Supports the collection of 13 network-related system calls and system metrics
- Enhances the analysis process with 6 correlation algorithms and 8 dashboards tailored for ransomware characterization

Cryptographic Ransomware

- **Malicious software** that **encrypts** victim’s data and demands a **ransom**
- New ransomware families are constantly appearing
- CRIBA allows the observation of **characteristic** ransomware behavior:
  - Traverses all victims’ directories
  - Rewrites victims’ files with encrypted data
  - Adds a new file extension to encrypted files
  - Leaves ransom notes to inform the victim
  - Has high CPU consumption due to encryption algorithms



CRIBA in Action

- Study with 5 different Linux cryptographic ransomware families

CRIBA in Action

- Study with 5 different Linux cryptographic ransomware families
- Total of 26 different observations regarding generic statistics, ransom notes creation, data access and encryption patterns, and evasion techniques

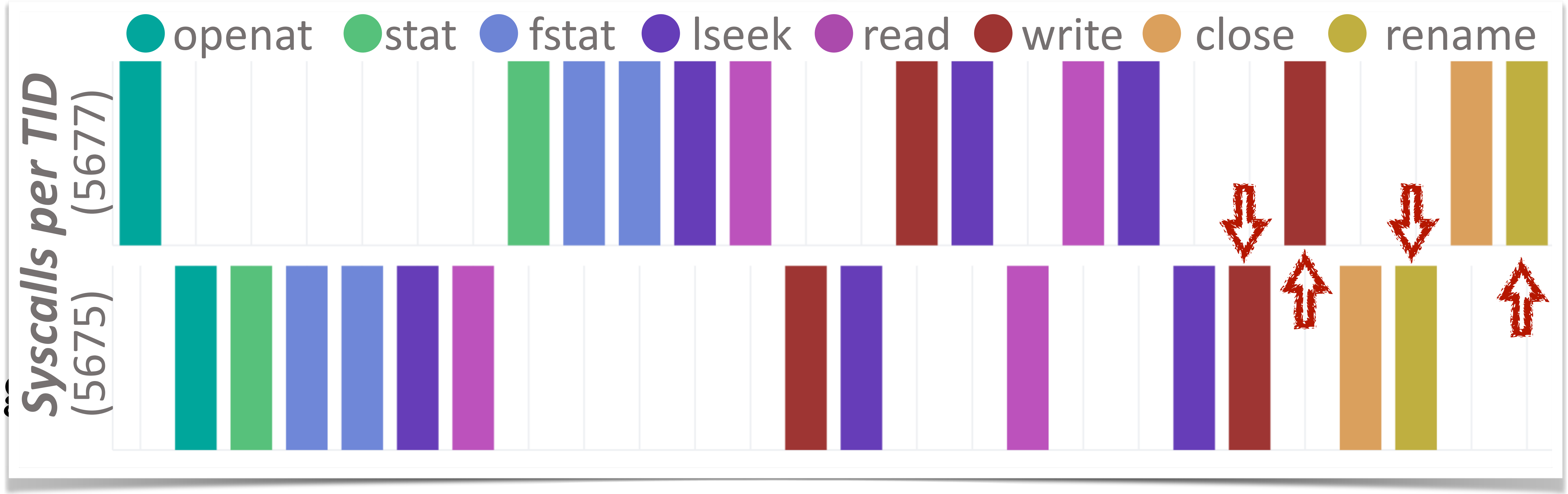
CRIBA in Action

- Study with 5 different Linux cryptographic ransomware families

- Total of 26 different observations regarding generic statistics, ransom notes creation, data access and encryption patterns, and evasion techniques

    - **Metadata-related** operations are the most predominant (lseek, stat, fstat)

CRIBA in Action



- Study with 5 different Linux cryptog

- Total of 26 different observations regarding generic statistics, ransom notes creation, data access and encryption patterns, and evasion techniques
  - **Metadata-related** operations are the most predominant (lseek, stat, fstat)
  - **Concurrent encryption actions** in RansomEXX that may lead to **data corruption**

CRIBA in Action

- Study with 5 different Linux cryptographic ransomware families
- Total of 26 different observations regarding generic statistics, ransom notes creation, data access and encryption patterns, and evasion techniques
  - **Metadata-related** operations are the most predominant (lseek, stat, fstat)
  - **Concurrent encryption actions** in RansomEXX that may lead to **data corruption**
  - **Different system call sequences** for file access (based on the targeted file)

CRIBA in Action

- Study with 5 different Linux cryptographic ransomw

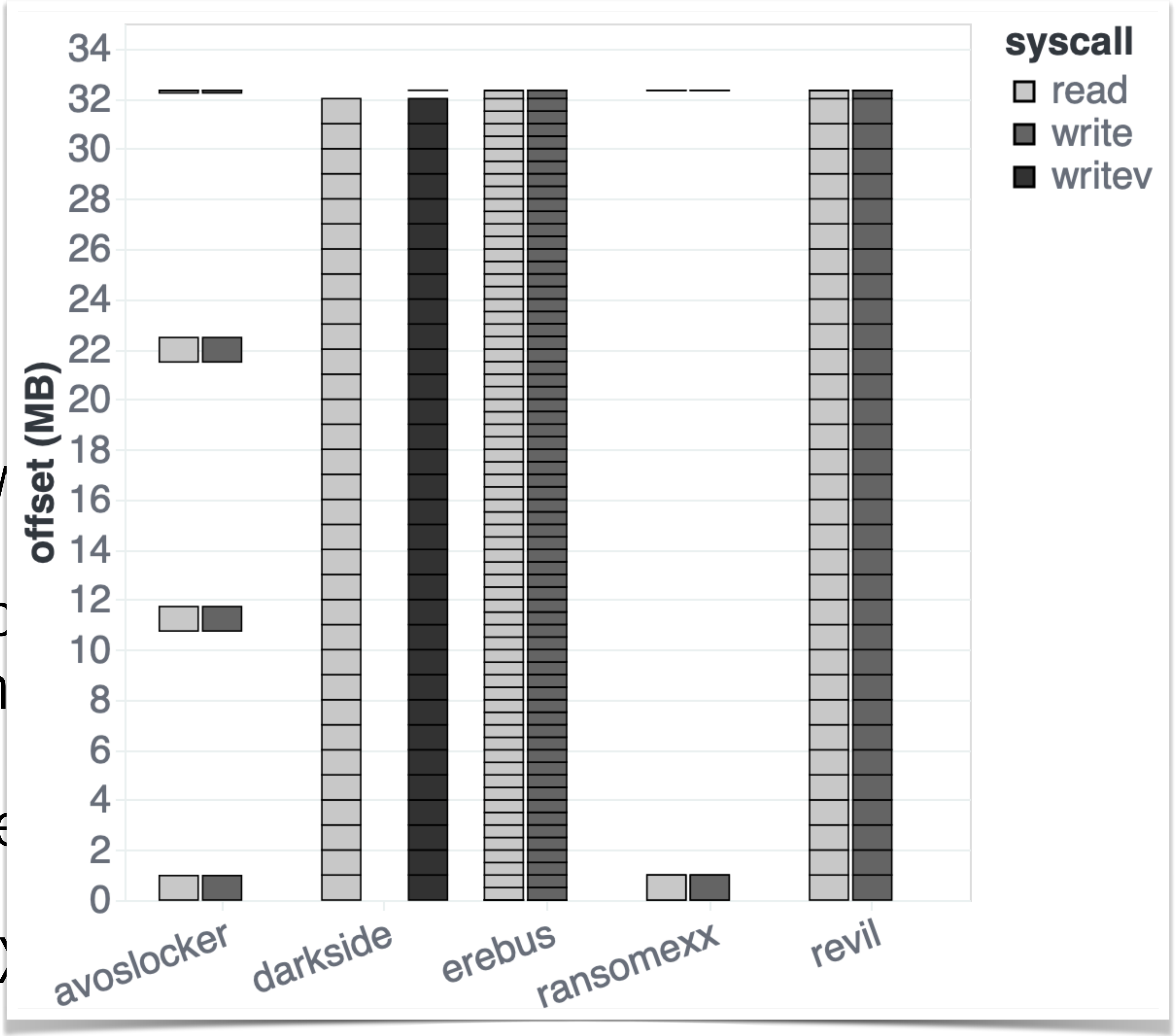
- Total of 26 different observations regarding generic access and encryption patterns, and evasion techn

  - **Metadata-related** operations are the most pre

  - **Concurrent encryption actions** in RANSOMEX

  - **Different system call sequences** for file access (based on the targeted file)

  - Some families process only **partial content** of files or target **specific file extensions**



CRIBA Summary

- CRIBA highlights **DIO’s usefulness** by extending it with tailored analysis and visualization
- Our study shows that **different features** must be considered for a **clear understanding** of ransomware’s **intrinsic** behavior
- This knowledge is key for improving detection tools for Linux cryptographic ransomware

Conclusion

- **CaT**, a framework for diagnosing storage and network I/O requests of **distributed systems**
  - Follows a content-aware approach that allows observing how data flows across components
  - Useful for uncovering data corruption and adulteration issues

- **DIO**, a generic tool for diagnosing **data-centric applications**’ storage I/O
  - Provides the flexibility to narrow or broaden the collection, analysis and visualization of I/O behaviors
  - Useful for debugging, validating and exploring both known and unknown storage patterns

- **CRIBA**, a practical tool for characterizing the I/O behavior of **cryptographic ransomware**
  - Automates the analysis and visualization of specific ransomware behaviors
  - Useful for better understand ransomware attacks and enhance detection tools

Publications

Core Publications

- **Tânia Esteves**, Francisco Neves, Rui Oliveira and João Paulo. **"CaT: Content-aware Tracing and Analysis of Distributed Systems"**. In *22nd International Middleware Conference*, 2021.
- **Tânia Esteves**, Ricardo Macedo, Rui Oliveira and João Paulo. **"Diagnosing Applications' I/O Behavior through System Call Observability"**. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2023.
- **Tânia Esteves**, Bruno Pereira, Rui Pedro Oliveira, João Marco and João Paulo. **"CRIBA: A Tool for Comprehensive Analysis of Cryptographic Ransomware's I/O Behavior"**. In *42nd Symposium on Reliable Distributed Systems*, 2023.
- **Tânia Esteves**, Ricardo Macedo, Rui Oliveira and João Paulo. **"Toward a Practical and Timely Diagnosis of Applications' I/O Behavior"**. In *IEEE Access*, 2023.

Complementary Publications

- Mariana Miranda, **Tânia Esteves**, Bernardo Portela and João Paulo. **"S2Dedup: SGX-enabled Secure Deduplication"**. In *14th ACM International Conference on Systems and Storage*, 2021.
- **Tânia Esteves**, Ricardo Macedo, Alberto Faria, Bernardo Portela, João Paulo, José Pereira and Danny Harnik. **"TrustFS: An SGX-enable Stackable File System Framework"**. In *38th International Symposium on Reliable Distributed Systems Workshops*, 2019.