A PILOT STUDY OF CROSS-SYSTEM FAILURES

BY

JIANYAN CHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Advisers:

Assistant Professor Tianyin Xu

ABSTRACT

Today's distributed system infrastructures usually consist of multiple systems that cooperate to deliver computing and storage services together. The reliability of distributed system infrastructures not only depends on the reliability of individual systems, but also on the reliability of the interactions among multiple systems. In fact, failures of system infrastructures are often caused by failures of interactions across multiple systems. We term such failures as cross-system failures.

In this thesis, we conduct a pilot study on cross-system failures on seven real-world systems that are commonly used to build distributed system infrastructures. We analyze the characteristics of real-world cross-system failures in four different dimensions: failure impacts, triggering events, root causes of the failures, and fixes of the root causes. Our goal is to provide insights to guide future research on resolving cross-system failures, and to shed light on the techniques that improve the reliability of cross-system interactions and distributed system infrastructures.

ACKNOWLEDGMENTS

I would like to express my gratitude to Assistant Professor Tianyin Xu for his recognition, guidance, and support over the past two years. The first time I met Tianyin was in a school admission interview. Even though, I applied the master of science program in computer science in University of Illinois Urbana-Champaign (UIUC), deep in my heart I had doubts on whether I was able to take the challenge. Tianyin's recognition boosted my confidence and granted me the opportunity in knowing system research in the past two years. Prior to the admission of UIUC, I was not sufficient in system research. Tianyin provided many opportunities in system research to me, guided me to learn in system research projects from the very beginning. This thesis would not be possible without him.

I also want to express my sincere thanks to Anna Karanika, Lilia Tang, and Jinghao Jia for their help and inspirations throughout the study.

I would like to express my sincere gratitude to Andrew Yoo, Professor Shuai Mu, Hsuan-Chi Kuo, Xudong Sun, Sam Cheng, Elaine Ang, Professor Owolabi Legunsen, Qingrong Chen, and Professor Dimitrios Skarlatos for their help and guidance in my prior research projects.

I would like to extend my sincere thanks to Wenyu Wang, Yifan Zhao, and Zirui Zhao for suggestions of superior restaurants around UIUC. Life would not be as memorable without them, or their cars.

Special thanks to Xudong Sun and Sam Cheng for the time spent in the 2019 winter. It was a happy time in research.

Special thanks to Yifan Zhao for his recommendation of Roland-P30. This electric piano helped me go through the lonely "quarantine" of COVID-19.

In addition, I would like to thank my parents for their wise counsel and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Nowadays, distributed system infrastructures are usually built with multiple systems. Different systems are specialized in providing different functionalities. They often rely on the functionalities of each other to provide storage and computing services. For example, a data-processing infrastructure typically consists of at least a data-processing framework (e.g., Spark [1, 2] or Flink [3, 4]), a distributed storage system (e.g., HDFS [5, 6, 7] or Alluxio [8, 9]), a scheduling system (e.g., YARN [10, 11] or Mesos [12, 13]), and a coordination system (e.g., etcd [14] or ZooKeeper [15, 16]). Distributed system infrastructures are orchestrated by interactions among multiple systems.

The reliability of distributed system infrastructures does not only depend on the reliability of individual systems, but it also depends on the reliability of system interactions. In reality, the interactions of multiple systems often cause severe consequences. For example, Google reported outages of several systems in December 2020. The outage was caused by insufficient space allocation for authentication services in the storage systems [17]. In the same month, Facebook experienced failures in the chat service in Messenger, Instagram, and Workplace services when the integration of Messenger and Instagram's messaging application infrastructure was under development [18].

Such failures manifest when there are errors in the system interactions. We term such failures *cross-system failures* since they only take place during interactions of multiple systems. That is to say, cross-system failures do not take place if only one system is involved.

Cross-system failures are challenging to be resolved because multiple systems are involved and their interactions are complex. This implies the complexity of failure characteristics and new challenges for reliability techniques. Most systems are developed with generality in mind by providing client API (application programming interface) for other systems. For example, YARN [11] provides standard interfaces for resource allocation requests from other systems. However, it is impractical for system developers to take care of all possible situations and edge cases that may occur when a system interacts with other systems. It is neither practical for system developers to test all possible systems that use the same client interface. Therefore, cross-system failures require specialized techniques.

However, we find that cross-system failures are not well-studied in literature. It is hard to find general failure characteristics such as failure impacts, triggering events, root causes, and how root causes are fixed. Due to the lack of understanding, it would be very challenging to develop effective and practical techniques that help resolve cross-system failures. In this thesis, our goal is to study the characteristics of cross-system failures. We are motivated to take the first step in understanding cross-system failures in a systematic and comprehensive manner. We aim to provide insights for future research on dealing with cross-system failures.

A cross-system failure can manifest when two systems are interacting with each other. System interactions are in the form of API invocations. The system that invokes the API functions is referred to as an *upstream system*. The system that provides the API functions is referred to as a *downstream system*. The upstream system invokes API functions of the downstream system and processes the returned messages. The upstream system may also communicate by saving data in the storage shared with the downstream system. In simple cases, failures can take place if the returned messages from the downstream system are unexpected by the upstream system. As shown in Figure 1.1, SPARK-27239 [19] happened when Spark incorrectly interpreted compressed data from HDFS. HDFS set the length of any compressed data to -1. Spark, without knowing that, asserted that the length of any data read from HDFS should be no smaller than 0. As a result, the assertion failed when Spark was reading compressed data from HDFS.

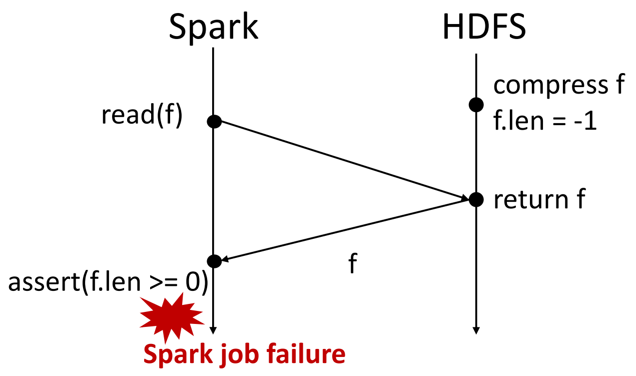


Figure 1.1: Spark assertion failed when checking the length of a compressed file from HDFS, causing job failures.

Cross-system failures may reflect more complex problems in interactions between the upstream and the downstream systems. Figure 1.2 illustrates FLINK-12342 [20], a failure caused by interactions between Flink and YARN. Flink relies on YARN for resource management and container allocation. Flink sends out all number of pending requests for container allocation to YARN at each heartbeat, and YARN allocates containers based on the numbers received from the heartbeat message. When a container is allocated, Flink decreases the number of pending requests by one to reduce the number of container allocation requests. However, when the allocation took much longer than the heartbeat interval, YARN accumulated the container requests from Flink. Eventually YARN was overwhelmed when allocating thousands of containers while Flink only needed 200. As a result, Flink could not start its streaming jobs, and YARN could not respond to other requests.

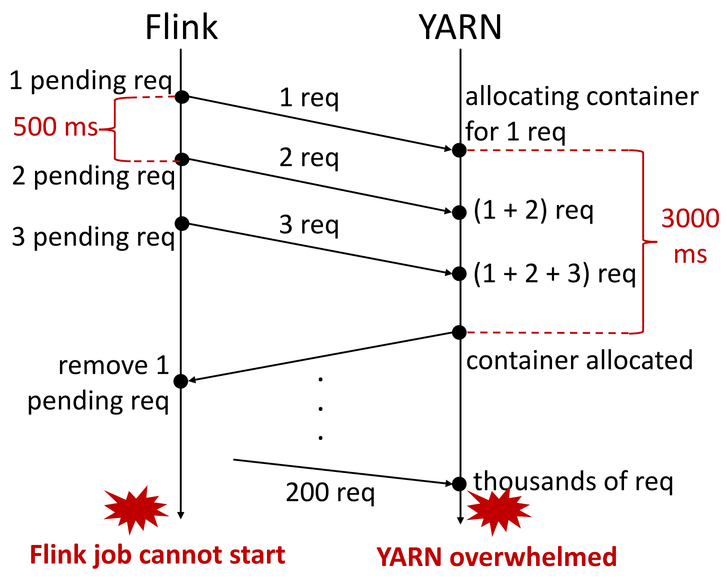


Figure 1.2: A failure caused by interactions between Flink and YARN, which finally overwhelmed YARN and prevented Flink from starting its jobs.

The rest of this thesis is structured as follows. Chapter 2 discusses the interaction interfaces among seven different systems, providing necessary context in understanding how upstream systems interact with downstream systems. Chapter 3 introduces rules for data collection and the analysis methodology for Chapters 4-7. Chapters 4-7 present the characteristics of cross-system failures concerning failure impacts, failure triggering, failure root causes, and fixes. Chapter 8 discusses related work and Chapter 9 concludes the thesis.

CHAPTER 2: INTERFACES AND INTERACTIONS

This chapter discusses the details of the interaction interfaces of 7 systems with a focus on answering the following questions:

- What do the interactions achieve?
- What are the programming interfaces for the interactions?
- Which programming interfaces are related to the control plane?
- Which programming interfaces are related to the data plane?

We in total studied 7 systems listed in Table 2.1:

System	**Desc.**
HBase	A distributed database.
Hive	A data warehouse with SQL like inteface.
HDFS	A distributed file system.
YARN	A distributed scheduling system.
Spark	A large-scale data processing system.
Flink	A distributed processing engine for data streaming.
Kafka	A distributed event streaming system.

Table 2.1: The systems we study for cross-system failures.

We choose the seven cloud systems for cross-system failure study because they are widely used and are usually used cooperatively in developing distributed system infrastructures.

Each system interacts with one or several other systems for achieving different tasks:

- Hive is used by Spark and Flink as a metadata storage and a HiveQL executor.
- Kafka is used by Spark and Flink as data streaming sources.
- YARN is used by Flink and Spark as a resource manager.
- HDFS is used by HBase and others as a file system and a storage backend.

There are other possible combinations, but limited by the issues we studied, we do not encounter issues relating to interactions between, such as, Flink and Spark, or Hive and YARN. Thus, their interaction interfaces are not discussed.

During our study, we observed mainly 4 types of interactions (Table 2.2):

- Metadata processing takes place when an upstream system saves metadata in a downstream system. The downstream system provides functions to manipulate metadata for the upstream system.
- Stream processing takes place when an upstream system requests data streams from a downstream system. The upstream system invokes APIs provided by the downstream system to establish connections and fetch data streams.
- Resource management takes place when an upstream system delegates job executions to another system. The upstream system invokes APIs provided by the downstream system mainly to allocate resources or modify resource requests for submitted jobs.
- File system: the upstream system uses the downstream system as a file system and a storage backend. The upstream system is able to issue file operations such as creating a file and renaming a file to the downstream system.

Types	**Upstream System**	**Downstream System**
Metadata Processing	Flink, Spark	Hive
Stream Processing	Flink, Spark	Kafka
Resource Management	Flink, Spark	YARN
File System	HBase, others	HDFS

Table 2.2: Type of interfaces studied for cross-system failures. Upstream systems invoke API provided by downstream systems. Flink and Spark mainly act as upstream systems.

Cross-system interfaces can be further divided into control-plane ones and data-plane ones based on whether they are interfaces for controlling data processing or handling data processing. We follow the definitions of control plane and data plane from Zamfir et al. [21]:

> "*The control plane of a datacenter application is the code that manages or controls data-flow and implements operations like locating a particular block in a distributed file system, maintaining replica consistency in a metadata server, or updating routing table entries in a software router.*"

> "*The data plane is the code that processes the data. Examples include code that computes the checksum of an HDFS file system block or code that searches for a string as part of a MapReduce job.*"

This chapter introduces each type of interfaces and discuss the control-plane and data-plane parts of the interfaces.

```
private[client] sealed abstract class Shim {
  protected def findMethod(klass: Class[_], name: String, args: Class[_]*): Method = {
    klass.getMethod(name, args: _*)
  }
}

private[client] class Shim_v0_12 extends Shim with Logging {
  private lazy val alterPartitionsMethod =
    findMethod(classOf[Hive],"alterPartitions",classOf[String],classOf[JList[Partition]])

  override def createPartitions(
      hive: Hive,database: String,tableName: String,parts: Seq[CatalogTablePartition],
      ignoreIfExists: Boolean): Unit = {
    val table = hive.getTable(database, tableName)
    parts.foreach { s =>
      val location = s.storage.locationUri.map(
        uri => new Path(table.getPath, new Path(uri))).orNull
      val params = if (s.parameters.nonEmpty) s.parameters.asJava else null
      val spec = s.spec.asJava
      if (hive.getPartition(table, spec, false) != null && ignoreIfExists) {
        // Ignore this partition since it already exists and ignoreIfExists == true
      } else {
        if (location == null && table.isView()) {
          throw QueryExecutionErrors.illegalLocationClauseForViewPartitionError()
        }
        createPartitionMethod.invoke(...)
      }
    }
  }
}
```

Figure 2.1: The base `HiveShim` in Spark. Spark may dynamically load Hive functions like `alterPartitionsMethod`, which utilizes reflection `getMethod` to invoke Hive API; or implements Spark logic by overwriting existing functions like `createPartitions`.

2.1 METADATA-PROCESSING INTERFACES

2.1.1 Spark and Hive

Spark supports many Hive features including HiveQL, User Defined Functions (UDF), and using Hive Metastore. Spark uses `HiveSession` to handle all Hive-related jobs including adaptation of different versions of Hive clients, reading and writing Hive tables, connecting to Hive Metastore, and supporting ORC format conversion (Hive ORC is a special data structure to store Hive data).

To support data manipulations for Hive tables, a specific class called `HiveShim` is used to invoke Hive APIs (Figure 2.1). To support different Hive versions, multiple `HiveShim` classes are used to address version differences (Figure 2.2). A base class is used to handle most of

```scala
private[client] class Shim_v0_13 extends Shim_v0_12 {
  override def createPartitions(
      hive: Hive,
      db: String,
      table: String,
      parts: Seq[CatalogTablePartition],
      ignoreIfExists: Boolean): Unit = {
    val addPartitionDesc = new AddPartitionDesc(db, table, ignoreIfExists)
    parts.zipWithIndex.foreach { case (s, i) =>
      addPartitionDesc.addPartition(
        s.spec.asJava, s.storage.locationUri.map(CatalogUtils.URIToString(_)).orNull)
      if (s.parameters.nonEmpty) {
        addPartitionDesc.getPartition(i).setPartParams(s.parameters.asJava)
      }
    }
    hive.createPartitions(addPartitionDesc)
  }
}
```

Figure 2.2: Different versions of Hive shims may have different implementations for the same function. Shim_v0_13 inherits other functions from Shim_v0_12 by extending Shim_v0_12. `createPartitions` has different implementations in v0_12 and v0_13.

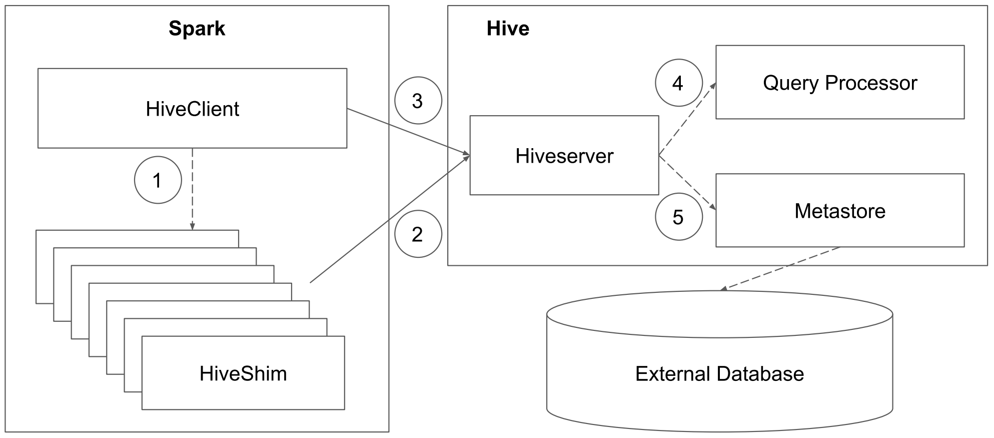


Figure 2.3: A simple workflow between Spark and Hive Metastore. The solid lines represent the interactions between Spark and Hive Metastore. The dash lines represent internal workflow within Spark and Hive.

the implementations for communicating with Hive. Many children classes are individually developed to address differences in different versions.

`HiveClientImpl` is the class that defines Spark's interactions with Hive. When an instance of `HiveClientImpl` is created, a configuration object is passed in to create a new configuration-specific Hive session. Meanwhile, `HiveClientImpl` matches the correct `HiveShim` version ①, provides wrappers for all the functions that create/read/update table schemas and table partitions ②, and for functions that submit HiveQL queries ③ to Hive (Figure 2.3). When a function is invoked, the wrapper function invokes the correct Hive API and waits for results. The HiveQL queries will be processed by Hive's query processor ④. Metadata-related requests will be processed by Hive Metastore ⑤. The invoked function is wrapped by a retry logic: when the number of function invocations is under retry limit or time limit is not approached, the function will be invoked again.

There are two other interactions between Spark and Hive: handling Apache Optimized Row Columnar (ORC) format and connecting to a secured Hive Metastore. ORC is a data storage format supported by Hive. Spark supports reading and writing ORC formatted Hive tables. The strategies to handle ORC tables are determined when a Hive session is created. The ORC handling strategies provide implementations to read and to write ORC files and tables. When Spark is connecting to a secured Hive Metastore, Spark periodically updates tokens to enable further API calls by scheduling the token update as a recurring job.

Reading and writing tables in Hive format and supporting different Hive versions are data-plane interactions because they either load or manipulate table metadata or interpret table metadata. control-plane interactions take place when Spark is connecting to Hive Metastore and when delegation tokens are involved for secured Hive Metastore connections. Spark maintains the connection with Hive Metastore to allow other actions and activities in the data plane.

2.1.2 Flink and Hive

Flink uses Hive for two purposes:

- Use Hive Metastore as a specific Catalog implementation for metadata.
- Grant Flink an alternative engine for reading and writing Hive tables.

Catalog is a Flink module that manages metadata for accessing data stored in a database or an external system. Hive Metastore is one type of the Flink Catalog implementations (others are Memory Catalog, Jdbc Catalog, and User-defined Catalog).

When Hive Metastore is chosen, Flink interacts with Hive Metastore by invoking functions defined in the class `HiveMetastoreClientWrapper`. The client wrapper class provides functions that create/read/update table schemas and table partitions (Figure 2.4). To manage different Hive versions, `HiveShim` provides different implementations for a specific func-

```java
public class HiveMetastoreClientWrapper implements AutoCloseable {
  ...
  public HiveMetastoreClientWrapper(HiveConf hiveConf, String hiveVersion) {
    this.hiveConf = Preconditions.checkNotNull(hiveConf, "HiveConf cannot be null");
    hiveShim = HiveShimLoader.loadHiveShim(hiveVersion);
    client = HiveCatalog.isEmbeddedMetastore(hiveConf) ? createMetastoreClient()
            : HiveMetaStoreClient.newSynchronizedClient(createMetastoreClient());
  }
  ...
  public void createDatabase(Database database){...}
  public void dropDatabase(String name, boolean deleteData, ...){...}
  public Database getDatabase(String name){...}
  public void createTable(Table table){...}
  public void dropTable(String databaseName, String tableName){...}
  public List<String> getAllTables(String databaseName){...}
  public Partition add_partition(Partition {...}partition)
  public boolean dropPartition(String databaseName, String tableName, ...){...}
  public Partition getPartition(String databaseName, ...){...}
  ...
}
```

Figure 2.4: `HiveMetastoreClientWrapper`, the class that defines Flink's interactions with Hive Metastore including manipulations of databases, tables, and partitions.

tion (Figure 2.5). When a `HiveMetastoreClientWrapper` instance is created, the correct `HiveShim` instance will be created to ensure the correct version of the function is invoked.

The reading and writing of Hive files and tables involve two phases: the storage phase and the runtime processing phase. At the storage phase, Hive data are stored in raw bytes using the minimum unit of Hive data in Flink, `Split`. Using reading as an example. When users decide to read Hive data, a reader object will be created to load data from the storage system as a collection of `Splits`. Meanwhile, the reader object is allocated with a `RuntimeProvider` object, which defines how to convert raw bytes to meaningful information based on metadata stored in the Hive Metastore. At the runtime, `Splits` will be distributed to Flink task executors that invoke functions in the `RuntimeProvider` and process the data (Figure 2.6).

Interactions defined in Flink catalogs are mainly in the data plane because they modify metadata for Hive tables. Interactions defined for reading and writing Hive files or tables are also data-plane interactions because they directly load or write Hive tables and files.

```java
/** Shim for Hive version 1.0.0. */
public class HiveShimV100 implements HiveShim {
  ...
  @Override
  public Class getHiveOutputFormatClass(Class outputFormatClz) {
    try {
      Class utilClass = HiveFileFormatUtils.class;
      Method utilMethod = utilClass.getDeclaredMethod(
        "getOutputFormatSubstitute", Class.class, boolean.class);
      Class res = (Class) utilMethod.invoke(null, outputFormatClz, false);
      ...
      return res;
    } catch (...) {...}
  }
  ...
}
/** Shim for Hive version 1.0.1. */
public class HiveShimV101 extends HiveShimV100 {}
/** Shim for Hive version 1.1.0. */
public class HiveShimV110 extends HiveShimV101 {
  @Override
  public Class getHiveOutputFormatClass(Class outputFormatClz) {
    try {
      Class utilClass = HiveFileFormatUtils.class;
      Method utilMethod =utilClass.getDeclaredMethod(
        "getOutputFormatSubstitute", Class.class);
      Class res = (Class) utilMethod.invoke(null, outputFormatClz);
      ...
      return res;
    } catch (...) {...}
  }
  ...
}
```

Figure 2.5: Different `HiveShim` classes that implement the function `getHiveOutputFormatClass` differently. In a more recent version, `HiveShimV110` provides a slightly different implementation of `getHiveOutputFormatClass` than `HiveShimV100` (older version) does.

2.2 STREAM-PROCESSING INTERFACES

2.2.1 Flink and Kafka

As a data processing engine, Flink is able to read data streams from multiple data-stream sources. Kafka is one of the data-stream sources Flink supports. Flink has modules called `connectors` for different data-stream sources. The Kafka connector is `FlinkKafkaConnector`.

The `FlinkKafkaConnector` is principally composed by two subclasses: a Kafka consumer, which is responsible for reading data from Kafka; and a Kafka producer, which is responsible for writing data to Kafka. Another important concept in Flink streaming is `stream barrier`.

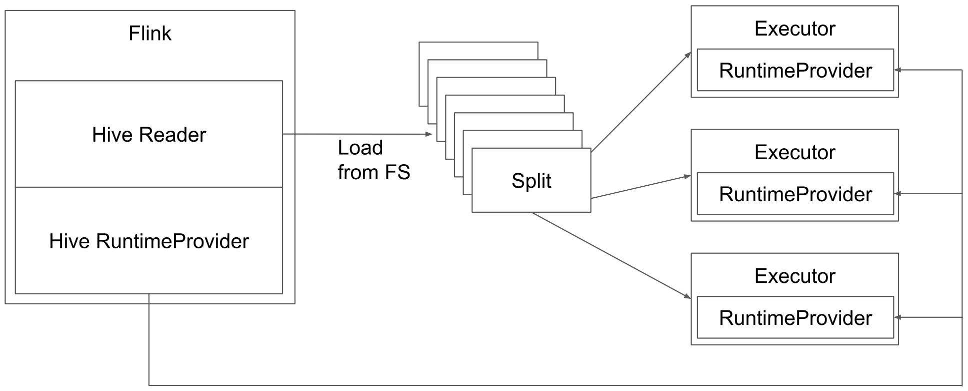


Figure 2.6: A simplified workflow for Flink reading Hive files and tables. Hive files and tables will be loaded and divided into multiple `Splits`, which are distributed to different executors. Flink's executors will be assigned with `Hive RuntimeProvider`, which has implementations for reading Hive files and tables from the `Splits`.

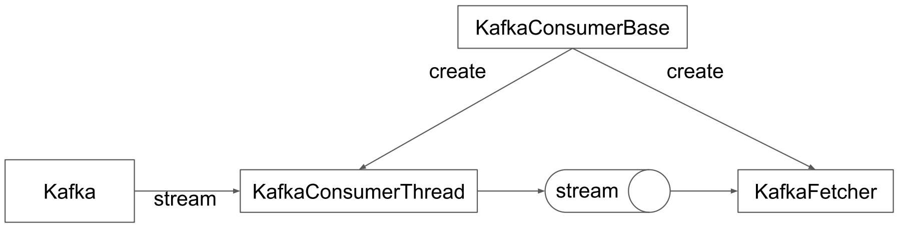


Figure 2.7: A simplified data stream fetching workflow for Kafka and Flink. `KafkaConsumerThread` fetches data streams from Kafka and pipes the data into a queue. `KafkaFetcher` converts data streams into Flink objects that will be processed later.

When Flink is fetching data streams from another system, `stream barrier` is also injected in the data streams. Each injected barrier represents a snapshot of the data fetching process. The Kafka consumer supports several configurable features:

- Fault tolerance. Flink is able to do checkpointing during data streaming with Kafka. By injecting `stream barrier` into the data streams and recording the number of barriers encountered during stream processing, Flink does checkpointing for a streaming job. If a job failure takes place, Flink is able to restore the streaming job from the latest checkpoint by starting reading from the data stream at the position labeled by the most recent `stream barrier`.

- Partition discovery. Kafka is a distributed event processing system. Events are partitioned by different Kafka topics. Different topics are handled by different groups of Kafka nodes. Flink is able to automatically discover topics and their associated group of nodes in Kafka even if Kafka topics are created dynamically. This is achieved by delegating partition discovery to a thread and periodically waking up the thread. Flink will dedicate job handlers for different Kafka topics when partition information is retrieved from Kafka by the thread.

Other features are not discussed because our dataset does not include related issues.

A simplified data stream fetching workflow (Figure 2.7) is as follows: when an instance of `FlinkKafkaConsumerBase` starts running, a thread is started to continuously polling Kafka messages by invoking Kafka consumer API. The polled Kafka byte messages, which are pushed to a queue, are later deserialized to Flink objects by `kafkaFetcher`.

In the fetching workflow, the control-plane interactions are maintained by an individual thread which constantly fetches raw byte messages from Kafka. Another thread handles the data-plane interactions which translates Kafka byte messages into Flink objects.

In partition discovery, the control-plane interactions are mainly invoking Kafka's consumer API for retrieving Kafka's partition information. The information is parsed by Kafka's API implementation. Flink only needs to loop through the returned object to access the information, which is a data-plane interaction. The control-plane and data-plane interactions take place synchronously in the same thread.

2.2.2 Spark and Kafka

Spark streaming has a simple model for receiving data streams from external data sources. The component `Receiver` receives external data streams and converts data streams to Resilient Distributed Datasets (RDD, the data unit in Spark) for data processing defined by users. Spark has a dedicated `Receiver` implementation for Kafka called `KafkaDataConsumer`. To interact with Kafka, `KafkaDataConsumer` invokes `KafkaConsumer` API to establish connections with and get records from Kafka (Figure 2.8).

The interactions between Spark and Kafka take place in both control plane and data plane. In the control plane, for example, Spark establishes connections with Kafka. Spark also provides wrapper function `seek` to access data at specific locations. On the data plane, for instance, Spark reads data streams from Kafka by invoking `poll`.

```
private[kafka010] sealed trait KafkaDataConsumer[K, V] {
  /** Reference to the internal implementation that this wrapper delegates to */
  def internalConsumer: InternalKafkaConsumer[K, V]
}
private[kafka010] class InternalKafkaConsumer[K, V](
    val topicPartition: TopicPartition,
    val kafkaParams: ju.Map[String, Object]) extends Logging {
    ...
  private val consumer = createConsumer
  private def createConsumer: KafkaConsumer[K, V] = {
    val updatedKafkaParams = KafkaConfigUpdater(...)....build()
    val c = new KafkaConsumer[K, V](updatedKafkaParams)
    ...
    c
  }
  def get(offset: Long, timeout: Long): ConsumerRecord[K, V] = {
    if (offset != nextOffset) {
      seek(offset)
      poll(timeout)
    }
    if (!buffer.hasNext()) {
      poll(timeout)
    }
    ...
  }
  private def seek(offset: Long): Unit = {
    consumer.seek(topicPartition, offset)
  }
  private def poll(timeout: Long): Unit = {
    val p = consumer.poll(Duration.ofMillis(timeout))
    val r = p.records(topicPartition)
    buffer = r.listIterator
  }
}
```

Figure 2.8: Example code pieces in Spark's `KafkaDataConsumer` that define interactions between Spark and Kafka for data streaming. `consumer` is the Kafka API object, `seek` and `poll` are the wrapper functions for Kafka's `consumer.seek` and `consumer.poll` APIs that locate the data streams and read the data streams.

2.3 RESOURCE-MANAGEMENT INTERFACES

2.3.1 Flink and YARN

Flink uses YARN as a resource manager for container allocations. Flink also deploys its `JobManager` and `TaskManager` instances on the containers oversaw by YARN. All interactions take place by invoking methods defined in the class `YarnResourceManagerDriver`. `YarnResourceManagerDriver` defines wrapper functions for `AMRMClient`, a YARN library

```
public class YarnResourceManagerDriver extends ...{
  private RegisterApplicationMasterResponse registerApplicationMaster() ...{
    ...
    return resourceManagerClient.registerApplicationMaster(...);
  }
  public void deregisterApplication(
    ApplicationStatus finalStatus, @Nullable String optionalDiagnostics) {
    final FinalApplicationStatus YarnStatus = getYarnStatus(finalStatus);
    final Optional<URL> historyServerURL = ...
    final String appTrackingUrl = historyServerURL.map(URL::toString).orElse("");
    try {
        resourceManagerClient.unregisterApplicationMaster(
                YarnStatus, optionalDiagnostics, appTrackingUrl);
    } catch (YarnException | IOException e) {...}
    ...
  }
  public CompletableFuture<YarnWorkerNode> requestResource(...) {
    checkInitialized();
    final CompletableFuture<YarnWorkerNode> requestResourceFuture = new
        CompletableFuture<>();
    ...
    resourceManagerClient.addContainerRequest(...);
    resourceManagerClient.setHeartbeatInterval(...);
    requestResourceFutures...
    return requestResourceFuture;
  }
  private void removeContainerRequest(AMRMClient.ContainerRequest pendingContainerRequest) {
    resourceManagerClient.removeContainerRequest(pendingContainerRequest);
  }
  private Collection<AMRMClient.ContainerRequest> getPendingRequests(...) {
    final List<AMRMClient.ContainerRequest> matchingRequests =
            resourceManagerClient.getMatchingRequests(...)
    return matchingRequests;
  }
  ...
}
```

Figure 2.9: Example code pieces in Flink's `YarnResourceManagerDriver` that define interactions between Flink and YARN for resource management.

that defines the actual implementation for interacting with YARN. Behind the scene, resource requests are sent to YARN periodically by a dedicated thread.

`YarnResourceManagerDriver` provides methods to start or stop applications, to make or cancel container requests, and to obtain container allocation information (Figure 2.9).

Interactions between Flink and YARN mostly take place in the control plane. By invoking the resource management API, Flink controls the resource requests that will be sent to YARN. Flink can also control the frequency of heartbeat messages. The data-plane part is

```
private def matchContainerToRequest(allocatedContainer: Container,
  location: String,containersToUse: ArrayBuffer[Container],
  remaining: ArrayBuffer[Container]): Unit = {
  ...
  val matchingRequests = amClient.getMatchingRequests(...)
  if (!matchingRequests.isEmpty) {
    val containerRequest = matchingRequests.get(0).iterator.next
    amClient.removeContainerRequest(containerRequest)
    containersToUse += allocatedContainer
  } else {
    remaining += allocatedContainer
  }
}
```

Figure 2.10: An example function in Spark's `YarnAllocator` that utilizes `AMRMClient` functions to implement resource management logic. `matchContainerToRequest` compares container requests of the given location against the allocated container. If the allocated container belongs to the given location, the corresponding container request is removed from `AMRMClient`.

finished by the YARN library `AMRMClient`, because `AMRMClient` processes heartbeat messages for any system that invokes the functions of it.

2.3.2 Spark and YARN

Similar to Flink, Spark uses YARN for resource management and job scheduling. Spark directly invokes `AMRMClient` functions in its own resource management logic. `YarnRMClient` implements functions for managing applications by invoking `registerApplicationMaster` and `unregisterApplicationMaster` of `AMRMClient`. `YarnAllocator` implements functions for requesting containers and decide how to use the allocated containers (Figure 2.10). By invoking `AMRMClient.addContainerRequest`, `AMRMClient.removeContainerRequest`, and `AMRMClient.getMatchingRequests`, Spark is able to add, remove, and query container allocations from YARN.

Interactions between Spark and YARN mostly take place in the control plane because `AMRMClient` processes heartbeat and request messages for any system that invokes it. Spark can only control what will be sent and access what is returned.

The interactions between Spark-YARN pair and Flink-YARN pairs are very similar because YARN provides a consistent resource management interface for other systems.

```java
public final class FSUtils {
  ...
  // Create a file.
  public static FSDataOutputStream create(FileSystem fs, Path path,
    FsPermission perm) throws IOException {
    if (fs instanceof HFileSystem) {
      ...
      if (backingFs instanceof DistributedFileSystem) {
        ...
        try
          return DistributedFileSystem.class
            .getDeclaredMethod("create", Path.class, FsPermission.class,...)
        } catch (...)
      }
    }
  }
  // Rename a file.
  public static void renameFile(FileSystem fs, Path src, Path dst) throws IOException {
    if (fs.exists(dst) && !fs.delete(dst, false)) {
      throw new IOException("Can not delete " + dst);
    }
    if (!fs.rename(src, dst)) {
      throw new IOException("Can not rename from " + src + " to " + dst);
    }
  }
  // Delete the region directory if it exists.
  public static boolean deleteRegionDir(final RegionInfo hri)
    throws IOException {
    ...
    return CommonFSUtils.deleteDirectory(fs,
      new Path(CommonFSUtils.getTableDir(rootDir, hri.getTable()), hri.getEncodedName()));
  }
  // Set file version by storing the version info in a file.
  public static void setVersion(FileSystem fs, Path rootdir, String version,
    int wait, int retries) throws IOException {
    Path tempVersionFile = new Path(rootdir, ...);
    FSDataOutputStream s = fs.create(tempVersionFile);
    try {
      s.write(toVersionByteArray(version))
      s.close()
    }
    ...
  }
  ...
}
```

Figure 2.11: Example functions in HBase's `FSUtil` that create a file, rename a file, delete a directory, and set file versions.

2.4 FILE-SYSTEM INTERFACES

HDFS provides a similar interface to be used as a file system for many other systems. HBase will be used as an example to illustrate the interfaces between HDFS and other systems.

Kafka and HDFS can also be integrated. However, in our cross-system failures dataset we do not encounter any Kafka-HDFS issues. Therefore, we did not study the interface between Kafka and HDFS.

HBase uses HDFS as a file system. They are heavily integrated. Most of the HDFS API calls take place in utility classes such as `FSUtils`, `RecoverLeaseFSUtils`, and some core HBase components like `HRegionFileSystem`.

`FSUtils` provides methods to interact with the underlying file systems such as creating a file, renaming a file, deleting directories, modifying file metadata like setting file versions, etc. (Figure 2.11). Note that HBase can also support other file systems.

The majority of the interactions between HBase and HDFS take place in the control plane because HBase issues file operations to the underlying file systems rather than directly processes the file data. The underlying file systems are insensitive to data formats and types because they save raw bytes into disk as a file. The underlying file system, HDFS, notifies HBase by returning messages or exceptions. Returning messages usually indicates success while exceptions indicates failure of the file operations.

CHAPTER 3: METHODOLOGY

3.1 DATASET

In total, we studied 98 cross-system failures in the 141 identified cross-system failures in seven different cloud systems: Spark [2], Flink [4], Hive [22], Kafka [23], YARN [11], HBase [24], and HDFS [7]. The seven systems are selected because they are commonly used together as component systems in distributed system infrastructures. All issues were collected from the Apache JIRA databases [25] with respect to the seven selected systems.

We used a heuristic while collecting the issues—*An issue in one system's JIRA database is cross-system-related if the related system's name is mentioned in the issue.* Besides this heuristic, other rules were also applied to filter issues in the database:

- Only select resolved issues because it is hard to draw conclusions for open issues.
- The reporter and the assignee of the issue should have different user IDs to exclude issues reported by the engineers (we target user-experienced failures).
- Only select "blocker," "critical," and "major" issues.

We collected 1145 raw issues and randomly sampled 450.

Heuristics cannot guarantee all issues selected are cross-system-related. For example, users may include data files stored in HDFS while using any of the selected systems. In user logs, "HDFS" may appear as a URI scheme while the issue is not cross-system-related. However, it is unreasonable to treat all such issues as non-cross-system issues because HDFS URIs may also appear in a cross-system issue.

We further conducted a consensus analysis by three group members individually. 450 issues are divided into 3 even sets and assigned to each group member. Each group member individually reviewed the assigned set of issues, and labeled each issue by "in-scope" and "out-of-scope". After the first round of revision, if an issue received two tickets of "in-scope", we believed it was a cross-system-related issue. Some issues might be difficult to be reviewed by an individual group member. All group members would review such issues together in group meetings to decide whether they were cross-system-related issues or not. In total, we identified 141 issues that described cross-system failures.

3.2 ANALYSIS METHODOLOGY

Failure impact analysis methodology. The impact of failures was summarized mainly from issue descriptions and messages from the issue comments. Issue reporters usually do not directly mention the impact of the failures in the issue descriptions. However, reporters often include the error messages, or failure targets when the failure took place. From the error message and the failure target, we were able to infer the failure impact. For example, in YARN-2790 [26], the issue description mentioned "NM log aggregation fail". From this message we learned that the logging service, which was a service of a system, failed. Reporter of YARN-2790 also showed some relevant error messages which included this sentence: "Skip log upload this time." From this message we inferred that YARN was able to handle the failure by skipping the logging service, which indicated that this failure did not result in an unexpected crash in YARN. Therefore, the specific failure was resolved as a partial failure that took place in YARN's system service.

A few issues do not include any useful information related to the impact of the failure. For example, in HBASE-1520 [27], the issue description only included one sentence: "In 'next' we are catching and ignoring IOExceptions - this is masking when we are having HDFS issues. We should throw the exception.", and there are only two comments: "this helps" and "+1", which obviously do not provide useful information to infer failure impacts. These issues are documented but not included in the failure impact analysis.

Failure triggering analysis methodology. Failure triggering analysis studies the complexity to trigger reported failures. Issue reporters usually show in the issue descriptions that under what situations failures take place. Sometimes, steps for failure reproduction are also discussed in the issue comments. This information is used to summarize the triggering events of the failures.

The optimal case is that reporters clearly documented steps and inputs to reproduce the failure. For instance, in issue SPARK-25206 [28], the reporter shows the exact sql queries to reproduce the reported failure (Figure 3.1). The triggering inputs can be directly summarized from the documented steps for reproducing the failure as follows:

1. File write from client, which corresponds to `write.parquet(...)`.
2. Database write from client, which corresponds to `CREATE TABLE ...`.
3. Database read from client, which corresponds to `select * from ...`.

More often reporters do not document the exact steps to reproduce the failure but describe the operations they did or actions they took before encountering the failure. In this situation,

```
spark.range(10).write.parquet("/tmp/data")
sql("DROP TABLE t")
sql("CREATE TABLE t (ID LONG) USING parquet LOCATION '/tmp/data'")
scala> sql("select * from t where id > 0").show
```

Figure 3.1: The steps to reproduce failure in the issue description of SPARK-25206 [28].

it is necessary to make logical inference about the necessary events for failure triggering. For example, in issue HIVE-3355 [29], the reporter used phrases such as "view the files" and "select the data" to describe the actions taken to trigger the failure. Therefore, parts of the triggering events could be summarized from the issue description. In the case of HIVE-3355, "select the data" is showing a client read event. However, logically we knew that the data needed to be in place before any client was able to read the data. Therefore, it was reasonable to infer that another necessary event was to write the data to the desired location.

An important note is that triggering events do not count the preparation steps to start a system. Operations such as compilation and modifying configuration files are not counted as triggering events. In other words, only events take place at the system runtime (after a system is initialized and started) are accounted for as triggering events. Some failures may manifest during system initialization, to avoid failures that can be triggered by zero events, we count that failures that manifest at system initialization need one event to trigger.

Root cause analysis methodology. Root cause analysis summarizes the root causes of the failures. Issue reporters usually do not know the root causes of the reported failures if they do not provide a patch. Developers sometimes provide help by identifying root causes and pointing to the correct fix. In issue YARN-8223 [30], the reporter only reported the actions taken before encountering the failure. However, the assigned developer directly mentioned the root cause of the failure in the first comment of the issue. Sometimes, developers may also summarize root causes in the corresponding pull request. From these developers' messages, we directly learned the root causes of failures.

There are also cases when developers point to a duplicated issue rather than explaining root causes. In such a case, we only analyze the duplicated issue and attribute the root cause of the duplicated issue to the current issue.

In the most extreme case, there are no root-cause-related messages documented in the issue nor the pull request. We need to review as much information as we can. The patch of the corresponding issue usually points to an accurate code pieces that cause the failure. Since we only select resolved issues, patches are available in all issues. Patches include code pieces that were removed and added. Reviewing such code pieces manifest clues of root causes. For example, in issue SPARK-15046 [31], the issue reporter documented a failure with a `NumberFormatException`. Developers did not directly mention the root cause in the comment or the corresponding pull request. The patch of the issue recorded that

```
- val renewalInterval = sparkConf.getLong(
-   "spark.YARN.token.renewal.interval", (24 hours).toMillis)
+ val renewalInterval = sparkConf.get(TOKEN_RENEWAL_INTERVAL).get
```

Figure 3.2: The patch of SPARK-15046 [31]

the variable `renewalInterval` was modified, the value of the parameter was obtained differently. We could therefore infer that the original way to access the configuration parameter `spark.YARN.token.renewal.interval` caused the failure. The difference between the old function and the new function to access `spark.YARN.token.renewal.interval` was that, the old function forcibly converted the value to `Long` type, while the new function did not do any conversion. This showed that the old function could avoid the documented `NumberFormatException`. Therefore, we were able to locate the root cause of this failure.

Fixes analysis methodology. Fixes analysis focuses on patterns of how bugs that cause cross-system failures are fixed. We conduct the analysis by summarizing the patch/patches of each issue. Patches do not only contain modifications on the source code. Oftentimes developers also modify tests in the same patch. We only study the modifications in the source code of a system. Some issues do not have any patches but only pointers to their duplicated issues. We treat the fixes of the duplicated issues as the fixes of the current issues.

3.3 THREATS TO VALIDITY

Real world characteristic studies are subject to a validity problem. We picked the seven systems because they are widely used in system infrastructures. However, most of the systems are implemented in Java, Spark is implemented in Scala, and Flink is mostly implemented in Java with some Scala implementations. Code patterns we observed are subject to the implementation languages of selected systems, and they may not be representative to systems implemented by other programming languages.

In issue selection, we used the heuristic: *the reporter and the assignee have different user IDs* to ensure the issues we study are problems encountered by users. However, it is hard to eliminate the cases if the issue reporter and the issue assignee are different while they are both engineers. This may potentially reduce the authenticity of cross-system failures if a portion of our datasets are developmental bugs.

When we were collecting issues, we did not set any quota on any system we picked. The issue distribution are reflecting the natural distribution of cross-system failures in the JIRA databases of the seven systems we picked. However, the distribution is skewed towards data-processing systems. In total, we studied 98 cross-system failures. Issues from Spark and Flink JIRA databases contribute to about 80% (77/98, 78.6%) of the issues. Therefore, we are more likely to encounter failures related to data processing. Our characteristic study for failure impact, failure triggering, root causes, and root cause fixes are subject to the biased issue collection.

CHAPTER 4: FAILURE IMPACTS

This chapter presents findings of impacts about cross-system failures. By summarizing the issue descriptions, we collected failure impacts for the studied cross-system failures.

We classify failure impacts of cross-system failures into three levels: system-level, task-level, and usability-level.

- At the system level, failures can have a system-wide impact. That is to say, users may be negatively impacted while accessing any system features. Runtime crash, performance degradation, and data loss are examples of system level consequences.
- At the task level, a subset of features a system supports become unavailable, but other features are still accessible by users. More specifically, when a system is processing tasks such as querying and data streaming, the task executions are not successful under specific settings and given specific input. However, the system is still able to perform the same type of tasks correctly with different settings or inputs and provide other services.
- At the usability level, requests submitted to a system give results. However, the results are wrong or uninterpretable to users. In this case, the failure impacts are dependent upon users' environments.

Impact Level	**#Issues**
System	36 (36.7%)
⊢ Whole-system failure	11 (30.6%)
⊢ Partial-system failure	25 (69.4%)
Task	45 (45.9%)
⊢ Task abort	40 (88.9%)
⊢ Task suspension	3 (6.7%)
⊢ Task restart failure	2 (4.4%)
Usability	15 (15.3%)
Not reported	2 (2%)
Total	98

Table 4.1: Statistics of cross-system failure impacts.

Impact Level	#Issues
Whole-system Failure	11 (30.6%)
⊢ Startup fails	5 (45.5%)
⊢ Runtime crash	5 (45.5%)
⊢ Performance degradation	1 (9%)
Partial-system Failure	25 (69.4%)
⊢ Job submission failure	11 (44.0%)
⊢ System service failure	6 (24.0%)
⊢ Resource leak	3 (12.0%)
⊢ Performance decline	3 (12.0%)
⊢ Data loss	1 (4.0%)
⊢ Job allocation failure	1 (4.0%)
Total	36

Table 4.2: Breakdown of system level impacts.

4.1 SYSTEM-LEVEL IMPACTS

Finding 4.1: 36/98 (36.7%) of cross-system failures have a global effect: 11/36 (30.6%) of system level impacts have crashing behavior, 25/36 (69.4%) are partial failures that decrease reliability or quality of service (Table 4.1).

Startup failure, runtime crashing, and system degradation are the main crashing behavior of cross-system failures (Table 4.2).

Startup failure is the impact when a system cannot be started with certain settings. YARN-9724 [32] reports a failure caused by missing support of federation in Spark. When YARN enabled federation, it invoked unimplemented functions for federation in Spark and triggered a `NotImplementedException`, which prevented the initialization of Spark.

Runtime crashing happens during system interactions when a system crashes and terminates. FLINK-3067 [33] reports a failure when Flink was streaming from Kafka and checkpointing the data streams using Zookeeper. A `RuntimeException` was thrown while confirming a checkpoint because Flink could not handle Kafka broker failures, which resulted in a crash of the streaming job. As a consequence, a restart of Flink cluster was required to resume the streaming job.

Performance degradation describes an issue that initially exhibits performance decline, but eventually the performance degradation will cause a system crash. One example is FLINK-12342 [20] as shown in Figure 1.2. When Flink was requesting containers from YARN, and if YARN took longer to allocate the containers than Flinks request timeout, Flink would mistakenly send the same requests again to YARN while YARN would treat the repeated requests as new requests. More requests meant heavier workload and slowed down YARN more, causing YARN not to meet Flinks timeout. As a result, Flink crashed with a request timeout and no jobs could be run.

Non-crashing behavior involves resource leak, system slowing down, system service failure, and job submission failures (Table 4.2).

Job submission failure is a special system level consequence. Certain system settings may prevent any job submissions, while in other settings, jobs can still be submitted and executed successfully. In SPARK-5164 [34], when users were submitting Spark jobs from a Windows Spark client to a Linux YARN cluster, the Spark jobs would fail because the Windows Spark client did not prepare the job environment with the correct Linux format. On the other hand, a Linux Spark client could successfully submit a Spark job to a Linux YARN cluster.

Systems have services like logging. Failure of certain services may not prevent systems from handling other tasks. In our cases, logging is a recurring service which is scheduled periodically by a system. If one of the recurrence fails to execute, the system may lose logs for all its activities but could still be able to finish other tasks or services. In YARN-2790 [26], a failure took place when YARN was writing logs to HDFS. To access HDFS services, YARN needed to provide a token. When the token expired, YARN did not update the token and failed to write logs to HDFS. However, YARN was still able to provide scheduling and resource management for other systems.

Resource leak may not have crashing behavior but systems could eventually become unavailable. In SPARK-18968 [35], temporary folders, created by YARN for each different Spark application in HDFS, were not deleted when Spark applications terminated. Users reported the failure as they observed abnormal storage usage in HDFS due to accumulating temporary files produced by Spark. Potentially, resource leak may also have crashing behavior if the HDFS disk is filled up with Spark temporary files which prevents YARN from launching new Spark application instances.

System performance decline is another non-crashing impact. This takes place when a system fails to correctly utilize resources. While the system is slowed down, jobs in the system can still work successfully. For example, in FLINK-8638 [36], when Flink failed to take checkpoints using HDFS due to HDFS disk problems, at an extreme case, Flink needed to recover from a recently completed checkpoint. However, recovery from checkpoints could lead to a significant delay, which reduced streaming throughput. Instead, users believed that Flink should allow a few checkpoint failures at the extreme scenarios before proceeding the recovery that slowed down the system.

Different from job submission failure, job allocation failure takes place after job submission, some jobs do not receive any resource allocation, therefore, those jobs cannot be executed.

Task	**#Issues**	**Examples**
SQL Operations	28 (66.7%)	create, insert, select, partition, sort by
Data Streaming	10 (23.8%)	DirectKafkaInputDStream, PersistentKafkaSource
Others	2 (9.5%)	Auxiliary Service, URL Validation
Total	40	

Table 4.3: Different tasks that fail with a task abort.

FLINK-4486 [37], reports a failure that certain Flink's job managers did not have resources allocated such that jobs assigned to those job managers could not be executed. The root cause of the failure was that resource allocation was performed before initialization of all job managers was finished. As a result, some job managers were not assigned with resources.

4.2 TASK-LEVEL IMPACTS

Finding 4.2: 45/98 (45.9%) of cross-system failure impacts are isolated in the task level.

Task-level impacts are the failures that prevent users from making specific requests or requests with specific workload. Nearly half of the failures have task-level impacts - 45.9% (Figure 4.1) of the issues we study result as unsuccessful execution of tasks.

4.2.1 Task Abort

The majority of task level impacts 40/45 (88.9%) involve task abort. Based on the systems we study, task abort mostly take place in two different types of tasks: SQL operations and data streaming (Table 4.3).

SQL Operations. This type of failures contribute to 66.7% of the task level failures (Table 4.3). When a system is handling SQL operations such as CREATE, INSERT, and SELECT, the process terminates unexpectedly with an exception.

For an SQL task, the query terminates with an exception without returning any results. For example, in HIVE-11166 [38], after users created a HBase table in Spark, users could not write data to the table with the `insert` statement. An exception was thrown and users were not able to complete the insertion.

Data Streaming. This type of failures contribute to 23.8% of the task level failures (Table 4.3). When a system is receiving and processing data streams from another system, the streaming job terminates unexpectedly with an exception.

For a streaming task, the streaming process terminates before data can be fully processed or data source is properly closed. SPARK-19361 [39] reports a failure between Spark and Kafka. Spark received data streams from Kafka and assumed that the offset of data streams should be contiguously increasing. In fact, the data stream offset pattern could change if the data stream was compacted. The streaming process terminated unexpectedly when Spark found data offsets did not follow a contiguous pattern. Yet, data streams from Kafka were not fully consumed.

Others. There are a few tasks that do not belong to SQL operations or streaming. Failure reported in YARN-8223 [30] prevented loading jar file stored in HDFS as auxiliary service (which allows users to pluggin other shuffling and sorting algorithms for MapReduce jobs) because jar files in HDFS were mis-treated as archived files (zipped or tar files). SPARK-1111 [40] shows a failure in validating HDFS URLs because the original use of Java URL class could not correctly recognize HDFS URLs.

4.2.2 Task Suspension

Occasionally, when tasks fail, they may not terminate. They may maintain an “executing” status but make no progress. SPARK-8374 [41] shows a failure between Spark and YARN. Spark uses YARN as a job scheduler which is able to schedule jobs based on job priorities. Given only two jobs A and B to be scheduled, where B has much higher priority than A, A should acquire all resources once B terminates. However, in this issue, A was resumed without acquiring any resources even when B was forcibly terminated. As a result, A appeared to hang and could not continue its execution.

4.2.3 Task Restart Failure

Systems are designed with failover mechanisms. When some tasks fail, a system may be able to recover the progress by restarting the job. By design, Flink is able to create savepoints during streaming jobs and resume from savepoints even if an accidental termination takes place. However, this resuming process may fail. In FLINK-3440 [42], Flink was receiving data streams from multiple Kafka partitions concurrently. However, Flink was only able to save the progress of one Kafka partition. When user restarted Flink, only one Kafka partition recovered the streaming progress. Data streaming for other partitions were started over from the beginning.

Most cross-system failures have implications on feature availability which prevent users from utilizing functionalities provided by both systems.

4.3 USABILITY-LEVEL IMPACTS

Finding 4.3: 15/98 (15.3%) of cross-system failures yield wrong or undesirable results that affect usability.

Usability-level impacts cause complications for user operations. However, the severity of failures depends on users' environments and requirements. In HIVE-3355 [29], the string "é" stored in HDFS and was correctly displayed if queried from HDFS. However, when querying the same string from Hive, the character "é" was displayed as "?" in the query result. This type of failures has no impact on system availability or functionality, but returns uninterpretable results to users.

CHAPTER 5: TRIGGERING EVENTS

This chapter presents the findings about triggering cross-system failures including triggering complexity in terms of number of triggering events and number of required logical nodes for triggering, and failure determinism.

Triggering complexity can be measured by number of events required for reproduction. We define an event as a user (or system) action or an external incident that triggers reactions of a software system. Examples for user actions can be read and write operations to database systems. Instances of system actions can be data stream production and checkpointing. Examples for external incidents can be I/O failures caused by hardware faults and system shutdown due to power failure. Our definition of event is consistent with the definition of event in prior work [43, 44, 45].

Triggering events were obtained by summarizing issue descriptions. Procedures required for starting each system and connecting different systems are not counted as triggering events because they are common for the same pair of systems.

Triggering complexity can be measured by number of required logical nodes for failure reproductions. By learning the root causes of and reproducing some issues, we learned the number of required nodes for different issues. For example, if any issue requires HDFS, at least 2 logical nodes are required because this is the minimum requirement for running HDFS. The issue description of each issue may also tell how many more nodes are required besides the minimum.

The failure determinism tells whether a failure can be reliably reproduced. By analyzing the root causes, we learned the sources of indeterminism of each issue. Issues we did not find indeterministic sources were identified as deterministic failures.

To better understand the issues, we mannually reproduced 8 failures successfully by following the instructions in the issues' descriptions. Some issues have patches that include tests that mock the triggering of cross-system failures. We did not reproduce failures by directly running new tests against older system implementations directly on a single system. Instead, we believe that a successful reproduction of cross-system failures need to make sure that systems are interacting. We manually set up systems and their connections to make sure that at least two systems were running and interacting.

Event Type	#%
File/database write from client	58 (59.2%)
File/database read from client	30 (30.6%)
Starting a client job	19 (19.4%)
Data stream creation	15 (15.3%)
Data stream consumption	9 (9.2%)
Shutdown a system	7 (7.1%)
Data stream production	6 (6.1%)
Starting a service	4 (4.1%)
I/O failure	3 (3.1%)
Checkpointing	3 (3.1%)
Stopping a client job	2 (2%)
Loading client files	2 (2%)
Other	11 (11.2%)

Table 5.1: Events that trigger cross-system failures. The % column reports the percentage of failures required by the event type to trigger the failure. Since there are many failures requiring more than one triggering event, the total percentage is greater than 100%.

Num. of events	#%	
1	43 (43.9%)	}Single event
2	31 (31.6%)	Multiple events: 56.1%
3	15 (15.3%)	
>= 4	9 (9.2%)	

Table 5.2: Minimum number of input events required to trigger the failures.

5.1 TRIGGERING COMPLEXITY

Finding 5.1: The majority 55/98 (56.1%) of the failures require more than one input event to manifest, 41% of them are associated with system specific features.

Table 5.1 shows the triggering events of cross-system failures. We consider events from a testing and diagnostic point of view. Event like “I/O failure” can be emulated by using testing tools. Events “File/database read from client” and “Data stream consumption” are similar that both events take place when an upstream system is reading data from a downstream system. However, the former is usually a one time event, the latter may take place whenever data streams are available. Therefore, they are treated as two different events; events “File/database write from client” and “Data stream production” are different in a similar manner. Figures 5.1 5.2 5.3 5.4 show example failures that require different numbers of triggering events.

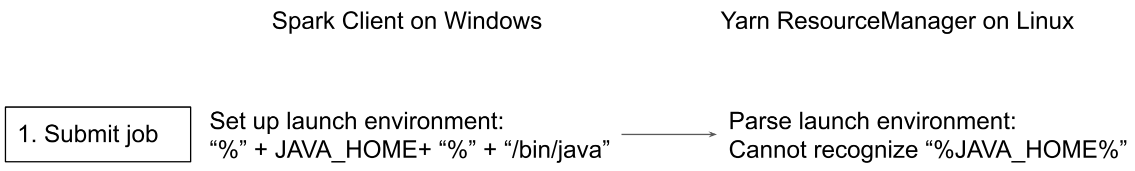

Spark Client on Windows
Yarn ResourceManager on Linux
1. Submit job
Set up launch environment:
"%" + JAVA_HOME+ "%" + "/bin/java"
Parse launch environment:
Cannot recognize "%JAVA_HOME%"

Figure 5.1: SPARK-5164 [34] where the failure was triggered by one event. When the user submitted a Spark job from a Windows machine to a YARN cluster on Linux machines, the job submission failed because YARN did not recognize the Windows file system delimiter "%".

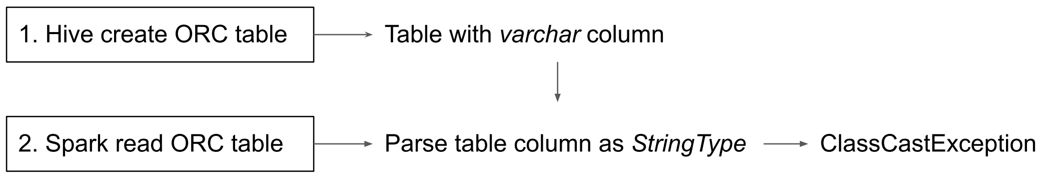


Figure 5.2: SPARK-18220 [46] where the failure was triggered by two events. (1) Users created an ORC table from Hive. The table needed to have columns with `varchar` type. (2) Users read the same table from Spark. `varchar` column type was not correctly propagated to Spark, and Spark treated `varchar` type as `StringType`, which caused a `ClassCastException`.

One of the complexities in triggering cross-system failures comes from system specific features. They can be functions and data types that are unique to a specific system, but they also participate in the interactions with other systems. `proctime` is a Flink specific data type. In FLINK-17189 [49], the failure manifested during the interaction between Flink and Hive Metastore because `proctime` was not properly supported in Hive Metastore. Therefore, when Flink was querying table content that contained `proctime`, an error showed up.

Finding 5.2: The ordering of events matters for all failures that need more than one input, but the events are mostly logically dependent.

Some events naturally take place before others, such as one cannot read without writing the data to the system first, and one cannot resume a system without stopping a system first. One typical example is SPARK-18220 [46] (Figure 5.2). The failure took place because Spark did not propagate data property when saving metadata at Hive Metastore for an ORC table. Therefore, when a query on the ORC table took place, the data could not be parsed correctly due to the missing data property. The failure manifested when users tried to read a table in ORC format. Yet, a table would not exist if users had not created the table in the first place.

These findings show the diversity and complexity of input events required to manifest cross-system failures. To expose failures during testing, we need to explore events from both

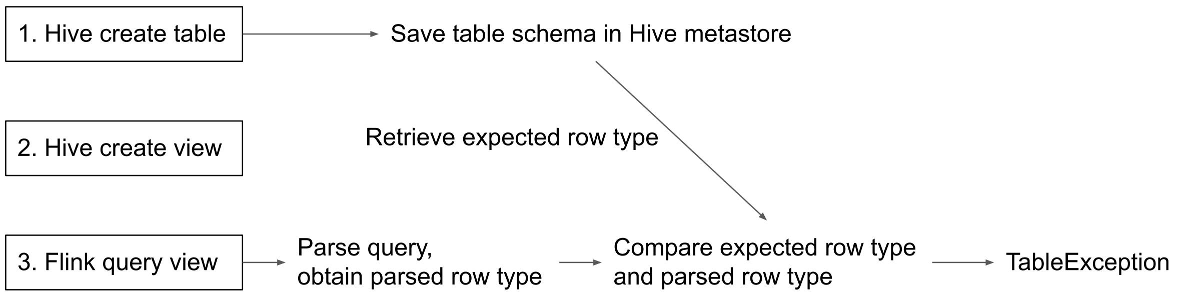

1. Hive create table
Save table schema in Hive metastore
2. Hive create view
Retrieve expected row type
3. Flink query view
Parse query, obtain parsed row type
Compare expected row type and parsed row type
TableException

Figure 5.3: FLINK-13197 [47] where the failure was triggered by three events. (1) Users created a table, the metadata of the table was saved in Hive Metastore. (2) Users created a view of the table. (3) Users queried the created view. Flink parsed the query string and obtained inferred row types of the target view. Meanwhile, Flink also obtained the expected row types from Hive Metastore. However, the inferred row type did not match the expected row type, which resulted in an error.

Num. of Logical nodes	**#%**	
1	18 (18.4%)	
2	10 (10.2%)	
3	16 (16.3%)	More than two logical nodes: 71.4%
4	21 (21.4%)	
>= 5	33 (33.7%)	

Table 5.3: Number of logical nodes required to reproduce the failures.

systems. While exploring events, it is beneficial to prioritize events that follow logical orders.

Finding 5.3: The majority 70/98 (71.4%) of cross-system failures require more than two logical nodes to reproduce (Details in Table 5.3).

Logical nodes are the Java processes required to reproduce the failure. Different systems have different minimum required Java processes to start the system. For example, HDFS requires minimally 2 Java processes to run. One for the namenode, and one for the datanode. On the other hand, executing HiveQL queries on Spark may only require one initial process that starts a spark terminal.

Cross-system failures take place between two systems. Given that some systems may require more than one process, it is common that triggering cross-system failures require more than two processes. This finding has implication in the complexity of cross-system reproduction in terms of automated testing. The more required logical nodes means more complexity in the testing automation.

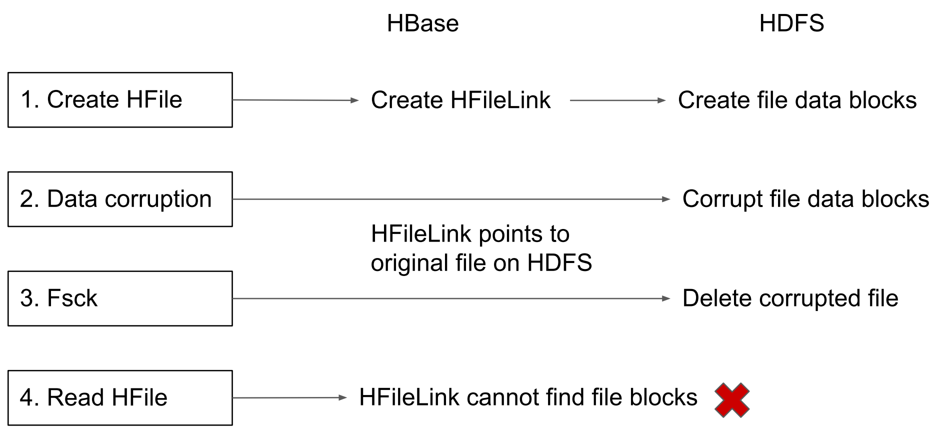


Figure 5.4: HBASE-16621 [48] where the failure was triggered by four events. (1) Users created a `HFile`, which created `HFileLink` in HBase and saved file data in HDFS. (2) When data corruption took place, the file data in HDFS were corrupted. (3) HDFS level fsck took place, which deleted the corrupted file in HDFS. However, `HFileLink` was still pointing at the deleted HDFS file. (4) When trying to access the `HFile` again from HBase, the `HFileLink` was not able to locate the file in HDFS because it had been deleted.

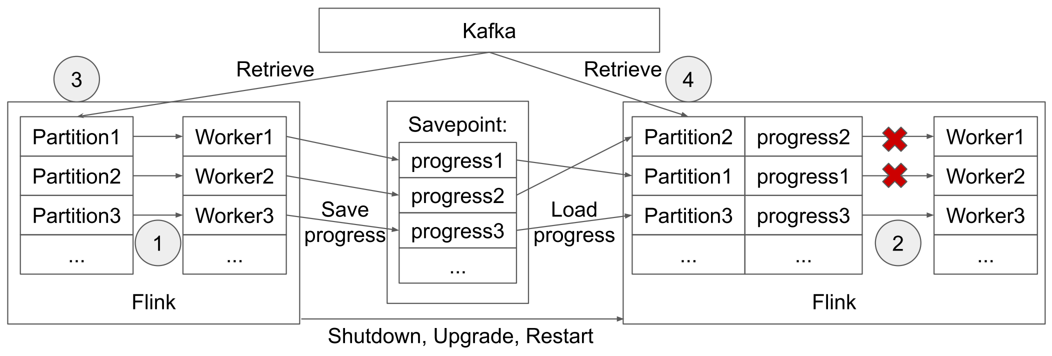


Figure 5.5: An illustration of the failure reported in FLINK-7143 [50]. The retrieved Kafka partitions were assigned based on the order of retrieved partition lists ① and ②. The order of the partition list was non-deterministic. If the order of partition list changed (as in ③ and ④), the failure manifested. Otherwise, failure did manifest.

5.2 FAILURE DETERMINISM

Finding 5.4: The majority 82/98 (83.7%) of the failures can be reproduced deterministically.

The majority of the cross-system failures can be reliability reproduced. We observed two sources of nondeterminism:

1. Timing of events. The majority of non-deterministic failures took place because they required special timing of triggering events.

2. Ordering of metadata. We observed one issue that could not be reliably reproduced if the ordering of metadata stayed the same.

Some triggering events may have random timing to other systems. Failure may only manifest if the triggering event takes place while one system is in a vulnerable state. For example, issue FLINK-17351 [51] reports a Flink failure caused by an unexpected Kafka failure or termination. Depending on when the Kafka failure or termination happened, Flink would react differently. If the Kafka failure took place when Flink was processing messages, Flink could recognize the failure and terminate the job properly. However, if the Kafka failure took place when Flink was checkpointing its current job progress, Flink would ignore the failure and was unable to finish or terminate checkpointing properly.

Ordering of metadata can be a non-deterministic source as shown in FLINK-7143 [50]. When Flink is streaming from Kafka, Flink retrieves metadata from Kafka about topics and partitions. However, the order of received topics and partitions are non-deterministic. Before FLINK-7143, Flink assigned workers based on the order received topics and partitions. Therefore, failures could arise as documented in issue FLINK-7143 (Figure 5.5). when a streaming job's progress was saved to a checkpoint, resuming from the checkpoint could fail. When the Flink job was resumed, Flink retrieved metadata from Kafka and tried to distribute topics and partitions to workers. However, the order of metadata changed, and some workers did not receive the same topics or partitions. Therefore, workers were not able to resume from previous checkpoints. However, the failure might be masked if the ordering of topics and partitions remained unchanged.

CHAPTER 6: ROOT CAUSES

This chapter presents the root causes of the studied cross-system failures. Software bugs are the major root causes of cross-system failures. As shown in Table 6.1, more than half (55/98, 55.1%) of root causes take place in the control plane. Nevertheless, root causes in the data plane are also significant (43/98, 45%), which corroborates with the findings in a recent study of Azure incidents [52]. Root causes of cross-system failures are discussed separately in the data plane and the control plane.

Bug Type	**#Issues**	**Descriptions**
Data plane	43 (43.9%)	
⊢ Table	25 (58.1%)	Inconsistent interpretations of table attributes
⊢ URL/path	7 (16.3%)	Misinterpretation of URL/path
⊢ Streaming	6 (14.0%)	Misinterpretation of streaming data
⊢ File	3 (7.0%)	Misinterpretation of file metadata
⊢ String	2 (4.7%)	Wrongly parsed strings
Control plane	55 (56.1%)	
⊢ Configuration	14 (25.4%)	Missing propagation of configuration objects and wrong configurations
⊢ System state	14 (25.4%)	Inconsistent state view and undesired state
⊢ Error handling	14 (25.4%)	Misinterpretation of errors
⊢ API misuse	10 (18.2%)	API misuse
⊢ Feature	3 (5.5%)	Non-supported features
Total	98	

Table 6.1: Summary of bug types and number of issues (corresponding to bug types) in the data plane and the control plane.

6.1 DATA-PLANE ROOT CAUSES

Finding 6.1: 43/98 (43.9%) of cross-system failures lie on the data plane.

When data are given to a system, data-plane code pieces process the data. Such data can be application-level content and metadata. Application-level contents could look like files, tables, data streams, etc. Metadata could be sizes, paths, and other attributes. When one system sends data to another, data may be interpreted differently by the receiver, which can break the interactions between the two systems. The majority of data-plane failures are caused by bugs in processing table attributes such as table schemas, table data types, etc. The rest of the failures in the data plane are caused by bugs in processing URLs/paths,

Information type	#Issues	Examples
Schema	10	ORC column names, column name letter cases
Data type	7	char, decimal, timestamp
Serialization, encoding	5	SequenceFileInputFormat, UTF-8, Shift_JIS
Other	3	nested column path, null argument
Total	25	

Table 6.2: Breakdown of table related data-plane issues.

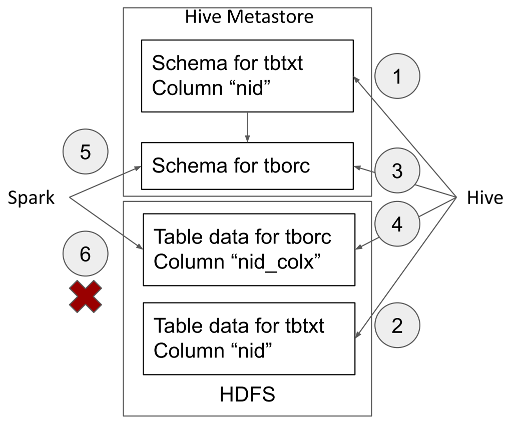
attributes related to stream processing, and files. We also observed a few failures caused by string-parsing bugs.

6.1.1 Table-Related Root Causes

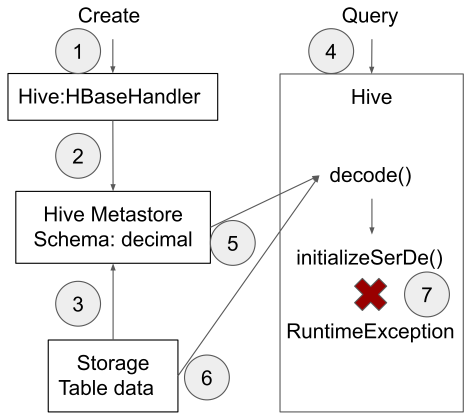
Three out of seven (Spark, Flink, and Hive) of our selected systems are closely related in processing table-related data. Spark and Flink support processing Hive-produced data, which are usually in the form of tables. We observe considerable number of the studied data-plane failures, 25/43 (58.1%), are caused by table-related bugs (Table 6.1).

Schema constitutes 10/25 (40%) of the table-related bugs (Table 6.2). A schema saves the metadata of a stored table. Schemas are usually stored separately from content in cloud systems. For example, Hive Metastore is an example metadata management system which is able to store table schemas for other cloud systems such as Spark and Flink. Different systems manipulate metadata in Hive Metastore to allow a consistent view of the same metadata. However, different systems may still interpret the same piece of metadata dissimilarly due to different implementations of their Hive Metastore clients. For instance, in SPARK-16605 [53] (Figure 6.1a), HiveQL and SparkSQL handled ORC column names differently. HiveQL added a specific suffix to ORC column names while SparkSQL did not know that. As a result, SparkSQL was not able to parse ORC tables created by HiveQL.

Data type constitutes 7/25 (28%) of table-related bugs (Table 6.2). Data type refers to the data types that are supported by both systems when reading the same table. To support the same data type, different systems need to have specific implementations for the data types. In database systems, there are usually hundreds of different data types if system specific features are also accounted. Different systems may handle same data types differently, which brings possibilities of oversights. In HIVE-17002 [54] (Figure 6.1b), when Hive was trying to access an HBase table, a `RuntimeException` occurred because Hive did not implement `decimal` data type operations.


Hive Metastore
Schema for tbtxt Column "nid"
Schema for tborc
Table data for tborc Column "nid_colx"
Table data for tbtxt Column "nid"
HDFS
Spark
Hive
1
2
3
4
5
6

(a) SPARK-16605 [53]



(b) HIVE-17002 [54]

Figure 6.1: **(a)** When Table `tbtxt` was created by Hive, table schema with column name "nid" was saved in Hive Metastore ①, and the table content was saved in HDFS ②. When Hive created the ORC version of `tbtxt`, `tborc`, the schema referenced the schema of `tbtxt` and was saved in Hive Metastore ③. However, Hive stored the table content in HDFS by appending column name "nid" with suffix "_colx" ④. When Spark queried `tborc`, Spark used the schema of `tborc` in Hive Metastore ⑤, which was referenced from the schema of `tbtxt` in Hive Metastore. Spark looked for the column name "nid" from the stored content in HDFS, which did not exist ⑥.
(b) Hive created an external table using `HBaseHandler` from HBase and saved table metadata in Hive Metastore ① ③. The table in HBase had columns with a `decimal` data type ②. When Hive tried to decode the table content, it retrieved schema information from Hive Metastore ⑤ and the table content from the storage system ⑥ to initialize `ObjectInspector` (Hive's internal data structure that represents different data types) for the `decimal` data type. However, the `decimal` data type's implementation of `ObjectInspector` was missing and a `RuntimeException` was thrown.

Serialization and encodings accounts for 5/25 (20%) of table-related bugs (Table 6.2). A failure takes place when different systems apply different serialization schemes or encodings to the same piece of content. For instance, in HIVE-3355 [29], "é" was saved in HDFS using UTF-8 encoding. When Hive client was operating on a system with Shift_JIS as the default encoding, Hive used it to parse "é", which resulted in a question mark "?".

Other table features are specifically related to Hive such as `nested column path`, which records locations of nested columns. Note that if other systems are included, failures may be caused by specific implementations of other systems.

6.1.2 URL/path-Related Root Causes

Misinterpretation of URL/path or the related resources contributes 7/43 (16.3%) of the data-plane failures (Table 6.1). A URL/path can be used across different systems to access a variety of resources. However, different systems may parse URLs/paths in a non-identical manner or treat the resources incorrectly, which causes failures. In SPARK-3685 [55], when users set HDFS URL, `hdfs:/tmp/foo`, as Spark's local directory, Spark failed to parse such said URL because the library Spark used, `java.io.File`, did not recognize the `hdfs` scheme.

6.1.3 Streaming-Related Root Causes

Misinterpretation of streaming data comprises 6/43 (14.0%) of data-plane failures (Table 6.1). Streaming is a task where one system produces data streams, and another receives data streams and processes the data. The receiving system may misinterpret the received data stream and fails to continue processing the data. For instance, SPARK-19361 [39] (Figure 6.2) describes a failure when Spark was streaming from Kafka. Spark assumed the data stream offsets sent by Kafka were always contiguous. However, when the topic is compacted in Kafka, the data stream offsets are not contiguous anymore. This failed the data stream validation at the Spark side and stopped the streaming operation.

6.1.4 File-Related Root Causes

Misinterpretation of file metadata involves 3/43 (7.0%) of the studied data-plane failures (Table 6.1). File metadata include file size and file location. Using file size as an example, SPARK-27239 [19] (Figure 1.1) describes a failure caused by the misinterpretation of a file size. HDFS sets a file size to -1 to represent that a file is compressed. However, when Spark

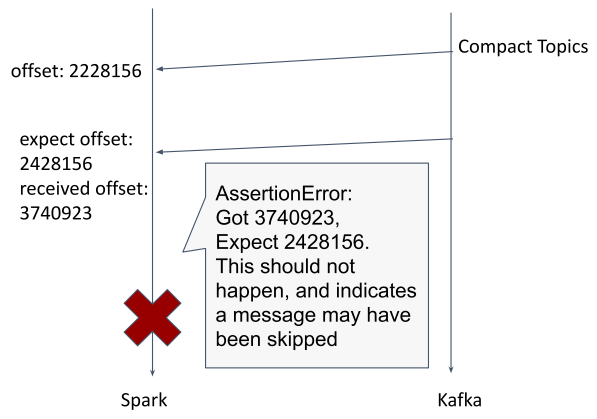

Compact Topics
offset: 2228156
expect offset: 2428156 received offset: 3740923
AssertionError: Got 3740923, Expect 2428156. This should not happen, and indicates a message may have been skipped
Spark
Kafka

Figure 6.2: SPARK-19361 [39]. When Spark was streaming from Kafka and the data streams were from compacted topics, Spark expected received offsets to be contiguous. At the moment when Spark received offset 2228156, Spark expected next offset to be 2428156 (an increment of 200000). However, offsets from compact topics did not have contiguously increasing offsets. Spark threw `AssertionError` when receiving the next offset, 3740923.

was processing such a file, Spark's validator asserted -1 as illegal, which caused a failure in file processing.

6.1.5 String-Related Root Causes

When strings are wrongly parsed, the information sent from one system to another cannot be correctly understood, which causes failures. We observed 2/43 such failures (Table 6.1). SPARK-4267 [56] describes a failure when Spark sent wrongly-parsed command line options to YARN. The root cause was that Spark erroneously parsed one command line option as three command line options, YARN failed at recognizing the second and third wrong options.

6.2 CONTROL-PLANE ROOT CAUSES

Finding 6.2: 55/98 (56.1%) of cross-system failures lie on the control plane, they primarily involve configurations, system states, and error handling.

Control-plane code pieces modify behavior of a system. Such code pieces change system behavior or affect behavior of other systems in various forms. We find five major types of bugs (configuration, system state, error handling, API misuse, and feature support) that cause cross-system failures (Table 6.1). Unlike data-plane failures, the root causes of control-plane failures are more evenly distributed among the different types of root causes. Each type of root causes is discussed with examples below.

6.2.1 Configuration

14/55 (25.4%) of the studied control-plane failures are caused by configuration-related bugs (Table 6.1). Configuration parameters can be used to coordinate system behavior. When incorrect configuration parameters are passed to a system, that system may be misconfigured and that may lead to failures. There are two reasons that cause incorrect configurations:

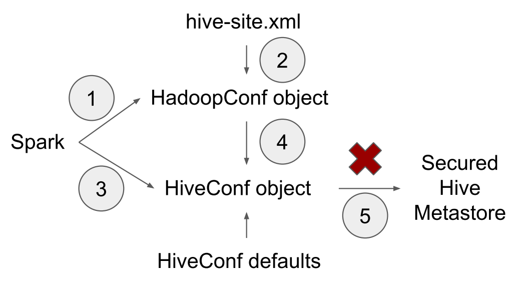
1. Missing propagation 10/14 (71.4%). The correct configurations are not propagated to the downstream system.

2. Wrong configuration 4/14 (28.6%). The wrong configurations are passed to the downstream system.

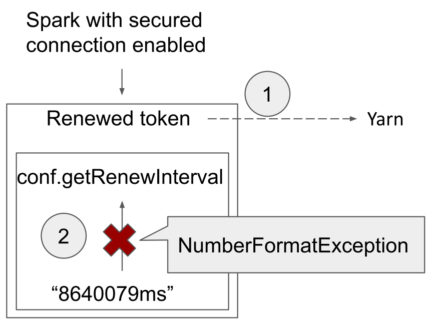
Missing propagation takes place because some configuration parameters are overwritten, or even the entire configuration object is replaced by a default one. For example, SPARK-17000 [57] (Figure 6.3a) reports a connection failure between Spark and Hive Metastore when the connection attempted to be secured. This takes place because a secured connection requires user settings on authentication information, but the object configuring Hive within Spark was not copied and user settings were omitted. The default configuration object was used; however, it did not have the correct configurations for secure connections. Therefore, secure connections between Spark and Hive Metastore could not be established.

Wrong configuration takes place when wrong parameters are passed to downstream systems. These are usually implementation bugs that set the configuration parameters to wrong values, or inconsistent configuration parameter changes in different versions of the same system. For instance, SPARK-15046 [31] (Figure 6.3b) describes a failure when Spark interacted with YARN through secured connections. Spark wrongly set a configuration parameter, which was used to renew the token for secured connections. An error was thrown when Spark was trying to access the parameter. Such an implementation mistake was masked because the parameter is only used when Spark is configured to interact with YARN through secured connections.

The aforementioned configuration-related root causes are different from parameter miconfigurations focused by prior work on configuration validation and misconfiguration detection [58, 59, 60, 61, 62]. Chen et al. [63] have discussed cross-system configuration dependencies; however, the work does not consider defects in code that handles configurations across the systems.



(a) SPARK-17000 [57]



(b) SPARK-15046 [31]

Figure 6.3: **(a)** A simplified workflow for Spark passing configuration objects to a secured Hive Metastore. Spark creates `HadoopConf` object ① and loads user configurations from `hive-site.xml` ②. When Spark connects to Hive Metastore, it creates a new `HiveConf` object ③ initialized with default values. However, Spark ignores user configurations stored in `HadoopConf` object ④ before trying to connect to the secured Hive Metastore. Without user parameters (authentication information), Spark is not able to connect to the secured Hive Metastore ⑤.
(b) A failure when Spark is renewing token required for secured connection with YARN. Spark needs a renewed token before connecting to YARN ①. Spark read from configuration object to learn the token renew interval ②, but encountered string "8640079ms", which cannot be parsed to numbers directly without stripping the "ms" suffix. Spark threw a `NumberFormatException` and was not able to renew tokens.

6.2.2 System State

14/55 (25.4%) of control-plane failures are caused by system-state-related bugs (Table 6.1). System state involves states shared between systems. The system states that are in charge of cross-system failures are generally managed through file I/O or system API. Sometimes, the changes of system states are not synchronized in both systems, which causes one system to operate in the wrong state. When two systems are cooperating, the state change of one system may lead to inconsistencies [64]. In our study, we observe two major patterns in faults related to internal system states:

- Incomplete view (10/14, 71.4%): the state of the downstream system is not known by the upstream system.
- Undesired state (4/14, 28.6%): the control flow leads to undesired state that eventually results in a failure.

Incomplete view contributes to 8/14 (57.1%) of system-state-related failures. Incomplete view between two systems takes place because one system takes actions before checking the state of the other system. For example, in HBASE-16621 [48] (Figure 5.4), a file system check was performed in HDFS, and files with missing data blocks were deleted. HDFS is a

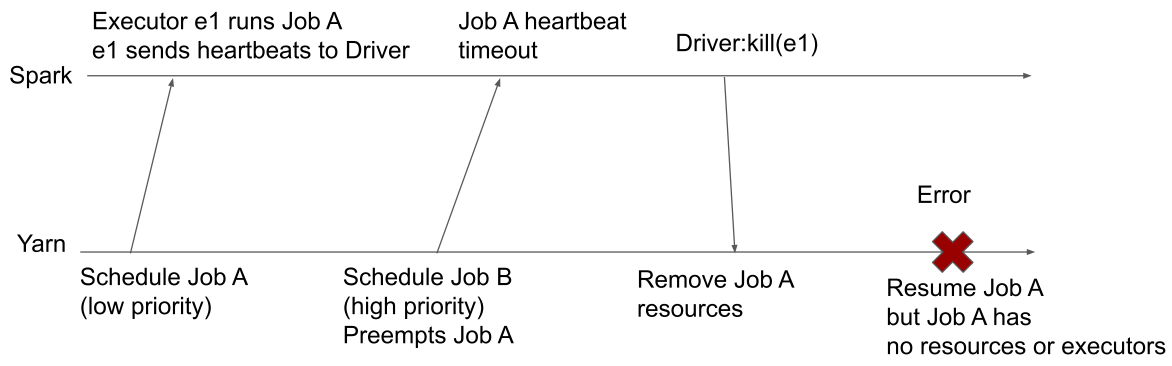

Executor e1 runs Job A
e1 sends heartbeats to Driver
Job A heartbeat timeout
Driver:kill(e1)
Spark
Error
Yarn
Schedule Job A (low priority)
Schedule Job B (high priority) Preempts Job A
Remove Job A resources
Resume Job A but Job A has no resources or executors

Figure 6.4: SPARK-8374 [41] describes a failure where a Spark job was resumed from preemption, the Spark job lost all resources. YARN first scheduled a low priority Job A. Job A executors maintained heartbeats connection with Spark driver to synchronize job status and other attributes. Then, YARN preempted the low priority job because there was a higher priority Job B submitted to YARN. Since Job A was preempted, the related heartbeat connection timed out, which triggered a function call in Spark that killed the executor for Job A. The execution kill triggered removal of resources for Job A in YARN. When Job A could be resumed, it lost all resources and executors and could not proceed back to its execution.

general system, it is not designed to explicitly communicate with a specific application using it. However, HBase still held `HFileLinks` to the deleted files, which led to failures in opening HBase regions because the accessed `HFileLinks` were pointing to invalid data blocks.

Undesired state contributes to 4/14 (28.6%) of system-state-related failures. In an upstream system, when methods are called or a variable is used to manipulate certain control flows, the downstream behavior is changed undesirably. For instance, SPARK-8374 [41] (Figure 6.4) shows a failure caused by the invocation of a method in the upstream system which led to undesirable state changes in the downstream system. When there was a heartbeat timeout between Spark driver and Spark job executors, Spark invoked `killExecutor`. The original intention was to cancel the executor with heartbeat timeout rather than stop the Spark job. However, the invocation of `killExecutor` removed the allocated resources at YARN's side for the corresponding job. As a result, the job failed to execute because it no longer had any resources.

6.2.3 Error Handling

14/55 (25.4%) of control-plane failures are caused by error-handling-related bugs (Table 6.1). Error-handling bugs refer to the misinterpretation of errors or exceptions sent by one system to another. When an upstream system communicates with a downstream system, the downstream system may throw errors or exceptions back to the upstream. If errors or exceptions cannot be correctly understood, the upstream system may fall into erroneous conditions. In FLINK-12342 [20], Flink misinterpreted the lack of acknowledgement from YARN as if it had lost the request, and so Flink resent requests including pending ones that had already been sent. YARN received the requests and continued processing them. Eventually, YARN was overwhelmed by the increasing container requests, and resources could be sufficiently allocated to Flink in time. As a result, Flink was not able to execute any jobs.

6.2.4 API Misuse

10/55 (18.2%) of control-plane failures are caused by API misuse (Table 6.1). In FLINK-9349 [65], Flink concurrently modified a linked list that was not designed for concurrent modification, which threw a `ConcurrentModificationException`. The linked list stored streaming metadata from Kafka. As a result, Flink could not continue the streaming job.

Additionally, we observe some (3/98, 3.1%) failures caused by features implemented by one system but not the other when two systems cooperate (Table 6.1). For example, YARN-9724 [32] describes a failure where YARN implemented federation, an HDFS feature that improves scalability and isolation, while Spark did not. As a result, Spark and YARN could not work collaboratively under federation mode.

CHAPTER 7: FIXES

This chapter presents how developers fixed bugs that caused the studied cross-system failures. For each failure, we studied the developer-provided patch that fixed the corresponding bug. Although cross-system failures are diverse, we find some common high-level patterns in fixing the root causes of the failures. The frequencies of the fix types with data-plane and control-plane root causes are summarized in Table 7.1. The majority of data-plane issues (35/43, 81.4%) were fixed by adding or improving checks and adding consistency in shared objects and system behavior. Control-plane issues were fixed in various ways given that numbers of each type of fixes in the control-plane issues are not drastically different as that of the data-plane issues. Fixes are discussed separately with respect to the data plane and the control plane.

Fixes	**#Issues**		
	Total	**Data Plane**	**Control Plane**
Add/improve checks	26 (26.5%)	18 (41.9%)	8 (14.5%)
Guarantee consistency	23 (23.5%)	17 (39.5%)	6 (10.9%)
Add propagation	12 (12.2%)	3 (7.0%)	9 (16.4%)
Add/improve error handling	11 (11.2%)	0 (0%)	11 (20.0%)
Add feature support	10 (10.2%)	4 (9.3%)	6 (10.9%)
Correct configurations	9 (9.2%)	0 (0%)	9 (16.4%)
Others	4 (4.1%)	0 (0%)	4 (7.3%)
No fix	3 (3.1%)	1 (2.3%)	2 (3.6%)
Total	98	43 (43.9%)	55 (56.1%)

Table 7.1: Statistics on how developers fixed the bugs that caused cross-system failures. The "data plane" and "control plane" columns show the number of issues in data plane and control plane respectively in terms of root causes.

7.1 ADD/IMPROVE CHECKS

Adding additional checks or conditionals is usually done by adding `if` conditions that handle the edge cases which cause failures. Improving checks is usually in the form of modifying the `if` conditions by using other functions, conditionals, etc.

```
- if (Utils.indexOf(path, "hdfs:/") == 0){
+ if (Utils.indexOf(path, "hdfs:/") == 0
+  || Utils.indexOf(path, "viewfs:/") == 0){
   Path remoteFile = new Path(path);
   ...
  }
```

(a) HIVE-19266 [66]

```
- long offset = getOffset(partitionID);
+ Long offset = getOffset(partitionID);
+ if(offset == null) {
+   continue;
+ }
  if (offset != ...){...}
```

(b) FLINK-3156 [67]

Figure 7.1: **(a)** Fix by adding additional check on whether the string object `path` was started with “viewfs:/”.
(b) Fix by adding additional check on whether the Long object `offset` was null.

7.1.1 Fixes in Data Plane

Finding 7.1: Nearly half (18/43, 41.9%) of the data-plane issues were fixed with the addition of a conditional. The majority of such issues (16/18, 88.9%) are related to metadata.

18/26 (69.2%) of the issues fixed by adding or improving checks were caused by data-plane bugs. These issues were fixed by adding or improving checks for omitted edge cases in metadata or content. The majority (16/18, 88.9%) of the checks are related to metadata. We observe metadata in various forms such as URL, data stream offset, and file format. Content refers to values of table columns and function arguments.

One system may omit certain URL forms of another system. For example, HIVE-19266 [66] describes a failure where Hive did not support URLs with a ViewFS scheme. The issue was fixed by adding an if-clause to check if a URL started with “viewfs” (Figure 7.1a).

Data stream offsets represent the progress of data streaming. Developers may omit the possible values for data stream offsets in edge conditions like when data streams are not yet available at the data stream source. FLINK-3156 [67] shows a failure caused by reading data stream offsets that were null. Shown in Figure 7.1b, the fix added a null check before the data stream offset was read.

When a system is downloading remote files, it is important to know the correct file format before the files can be correctly loaded. FLINK-15194 [68] describes a failure when Flink downloaded a zipped file from HDFS. The zipped file was not unzipped after downloading, which caused a `RuntimeException` when Flink tried to access the file. The bug was fixed by additionally checking whether the downloaded files were zipped or not (Figure 7.2a).

Similar to omitted edge cases in metadata, developers may omit edge cases when parsing data content. For example, FLINK-15429 [69] shows a failure when Flink read Hive tables that contained null timestamps. However, null timestamps could not pass Flink’s timestamp validation, which threw a `NullPointerException`. The fix added a null check for null timestamps (Figure 7.2b). If timestamps were null, null would be returned.

```
public Path asyncCopy() {
  copy(filePath, cachedPath);
+ if (isZipped) {
+   return expandDirectory(cachedPath);
+ }
  return cachedPath;
}
```

(a) FLINK-15194 [68]

```
public Object toHiveTimestamp(
  Object flinkTimestamp) {
+ if (flinkTimestamp == null)
+   return null;
  validate(flinkTimestamp);
  return flinkTimestamp;
}
```

(b) FLINK-15429 [69]

Figure 7.2: **(a)** The root cause was that the zipped files were not handled accordingly. This was fixed by an additional check on whether the target file was zipped or not. The method `expandDirectory` traverses all directories of a zipped file and extracts the file contents from source file system to target file system.
(b) A null `flinkTimestamp` caused a validation error. This was fixed by an additional null check on `flinkTimestamp`.

```
protected JobSubmissionResult
    submitJob(...){
+  waitForClusterToBeReady();
   ...
}
+ public void waitForClusterToBeReady() {
+  while (!ready)
+    Thread.sleep(200);
+ }
```

(a) FLINK-4486 [37]

```
private static ClientProtocol createProxy(
  final ClientProtocol cp,...)
  ...
- Object res = method.invoke(cp, args);
+ if (cp instanceof Closeable) {
+   ((Closeable)cp).close();
+ } else {
+   RPC.stopProxy(cp);
+ }
```

(b) HBASE-10029 [70]

Figure 7.3: **(a)** Method `waitForClusterToBeReady()` proactively checks whether the YARN cluster is fully initialized. The fix ensured that a job's submission would take place after cluster initialization completed.
(b) `ClientProtocol` is the interface implemented by HDFS proxies. The fix added extra checks on the interfaces implemented by the proxy object `cp` to ensure the method to be invoked existed.

7.1.2 Fixes in Control Plane

8/26 (30.8%) of the issues fixed by adding or improving checks are caused by control-plane-related bugs. These issues are mostly caused by (4/8) system-state-related faults and (3/8) API misuse. The fixes in the control plane enforce cooperations between two systems.

Adding checks enforces cooperations between two systems by correctly respecting shared system states. For example, FLINK-4486 [37] reports a failure caused by Flink submitting resource requests before the YARN cluster was ready for scheduling services. The fix checked and waited until YARN's cluster had finished its initialization as shown in Figure 7.3a.

Adding checks may also improve system cooperations by invoking the correct API. HBASE-10029 [70] describes a failure where HBase invoked a non-existing method for an HDFS proxy object. The fix checked if the existence of the method before invoking it (Figure 7.3b). Otherwise, another method would be invoked to achieve a similar result.

7.2 GUARANTEE CONSISTENCY

24/98 (23.5%) of the issues were fixed by ensuring consistency of data objects shared by multiple systems, and in system implementations that match behavior of other systems. Such fixes maintained consistent interpretations of communicated objects/data (such as table schema and command line argument strings) to allow correct data processing, or modified system implementations to better handle data processing or responses from other systems.
Finding 7.2: 17/43 (39.5%) of data-plane issues were fixed by maintaining a consistent interpretation between multiple systems. The majority of such issues (12/17, 70.6%) were fixed by modifying communication objects.

7.2.1 Fixes in Data Plane

17/23 (73.9%) of the issues fixed by guaranteeing consistency between multiple systems were data-plane issues.

Consistency fixes in the data plane were usually centered around modifications of communicated objects (12/17, 70.6%). The communicated objects involve table schemas (10/17), and command line argument strings (2/17).

Table schemas are usually saved in a centralized manner, For example, Hive Metastore manages metadata for other systems, so that different systems can have a consistent view on the content of the same tables. That is to say, table schemas are accessed by different systems in order to correctly access the table content. SPARK-21686 [71] describes a failure caused by the inconsistency in table schemas between Spark and Hive (same as reported in SPARK-16605 [53]) where Hive Metastore added a suffix to each column name in an ORC table. Nevertheless, Spark used column names without the suffix to read the tables. The fix added logic that appends suffixes to column names in Spark's table schemas when Spark was processing ORC tables (Figure 7.4).

A system may pass uninterpretable command line argument strings to another. For example, in SPARK-4267 [56], Spark removed quotation marks around arguments used for submitting jobs to YARN. The arguments were separated by spaces. Without the quotation marks, the arguments were treated as several invalid command line options by YARN (as discussed in Section 6.1.5). The fix respected quotation marks in arguments (Figure 7.5).

```
- val (fieldRefs, fieldOrdinals) = dataSchema.zipWithIndex.map {
-   case (field, ordinal) => oi.getStructFieldRef(field.name) -> ordinal
+ val (fieldRefs, fieldOrdinals) = requiredSchema.zipWithIndex.map {
+   case (field, ordinal) =>
+     var ref = oi.getStructFieldRef(field.name)
+     if (ref == null) {
+       val maybeIndex = dataSchema.getFieldIndex(field.name)
+       if (maybeIndex.isDefined) {
+         ref = oi.getStructFieldRef("_col" + maybeIndex.get)
+       }
+     }
    ref -> ordinal
  }
```

Figure 7.4: The patch in SPARK-21686 [71] shows that column names were directly obtained from the `dataSchema` object. The fix implemented a consistent view of `dataSchema` by adding the correct suffixes "_col" and `maybeIndex.get` to column names.

```
- sparkConf.getOption("spark.driver.extraJavaOptions")
+ val driverOpts = sparkConf.getOption("spark.driver.extraJavaOptions")
    .orElse(sys.env.get("SPARK_JAVA_OPTS"))
-   .foreach(opts => javaOpts += opts)
+ driverOpts.foreach { opts =>
+   javaOpts ++= Utils.splitCommandString(opts).map(YarnSparkHadoopUtil.escapeForShell)
+ }
```

Figure 7.5: The fix of SPARK-4267 [56] invoked `splitCommandString` which was able to parse argument strings like "one two three" as one option. Before the fix, the string "one two three" was parsed as three options, where "two" and "three" were invalid options for YARN.

The remaining (5/17, 29.4%) consistency fixes were modifications of system implementations by matching table format (3/5), file paths (1/5), and string encodings (1/5). Using table format as an example, HIVE-11166 [38] reports a failure caused by inconsistency in table format where the output format of HBase handler, `HiveHBaseTableOutputFormat`, could not be cast to `HiveOutputFormat`. The fix modified the implemented interface of the HBase handler to match the required output format (Figure 7.6).

```
-  static private class MyRecordWriter implements
-    org.apache.hadoop.mapred.RecordWriter<ImmutableBytesWritable, Object>
+  static private class MyRecordWriter implements
+    org.apache.hadoop.mapred.RecordWriter<ImmutableBytesWritable, Object>,
+    org.apache.hadoop.hive.ql.exec.FileSinkOperator.RecordWriter
```

Figure 7.6: HIVE-11166 [38] was fixed by implementing the interface that provided the correct output format.

7.2.2 Fixes in the Control Plane

6/23 (26.1%) control-plane issues fixed by ensuring consistency in system behavior that matches changes in another system.

In the control plane, consistency fixes were centered around system modifications. HBASE-16621 [48], discussed in Section 6.2.2, showcases how HBase and HDFS could have different understandings of the status of the same file. The fix had HBase perform its own file checking when restarting (Figure 7.7).

```
+ /**
+  * Scan all the store file names to find any lingering HFileLink files,
+  * which refer to some non-exiting files. If "fix" option is enabled,
+  * any lingering HFileLink file will be sidelined if found.
+  */
+ private void offlineHFileLinkRepair() throws IOException, InterruptedException {...}
```

Figure 7.7: HBASE-16621 [48] was fixed by adding offline checks for lingering `HFileLink` files.

7.3 ADD PROPAGATION

12/98 (12.2%) of the issues were fixed by adding the propagation of metadata or objects because the required information was not propagated to the destination when two systems were communicating.

3/12 (25%) data-plane issues were fixed by adding the propagation of metadata like data types. In FLINK-9384 [72], the type returned by a data source object was different from the type of data streams converted by the `deserializer`, which threw an exception when Flink was streaming from Kafka. The fix propagated type information to the `deserializer` which allowed it to convert data streams to the correct type (Figure 7.8).

9/12 (75%) of the issues fixed by adding propagation are control-plane issues. Most issues (7/9) were caused by missing propagation of configuration objects. For instance, HIVE-11250 [73] shows a failure of realizing a configuration change at runtime because the runtime changes at Hive were not propagated to Spark. This took place because Hive forgot to set the configuration update flag when propagating the new configuration. The fix was intuitive which was setting the update flag in the configuration class constructor (Figure 7.9a).

```
@Override
protected AvroRowDeserializationSchema getDeserializationSchema() {
-    return new AvroRowDeserializationSchema(avroRecordClass);
+    return new AvroRowDeserializationSchema(avroRecordClass,
   tableSchemaToReturnType(schema));
+  }
+
+  /** Converts the table schema into the return type. */
+  private static RowTypeInfo tableSchemaToReturnType(TableSchema tableSchema) {
+    return new RowTypeInfo(tableSchema.getTypes(), tableSchema.getColumnNames());
  }
+  public AvroRowDeserializationSchema(Class<? extends SpecificRecord> recordClazz,
   TypeInformation<Row> typeInfo) {...}
+  @Override
+  public TypeInformation<Row> getProducedType() {
+    return typeInfo;
+  }
```

Figure 7.8: Patch of FLINK-9384 [72] passed type information to `AvroRowDeserializationSchema`. When table content was deserialized, the correct type could be applied.

7.4 ADD/IMPROVE ERROR HANDLING

11/98 (11.2%) of the issues were fixed by adding or improving error handling code. Most issues (11/11, 100%) fixed this way were control-plane issues. (Note that we did not observe such bug fixes in the data plane, this does not mean that such bug fixes cannot take place in the data plane). A most commonly known form of errors is *exception*. They can be handled by `try/catch` statements. However, different systems may have other ways to handle errors, such that error-handling code pieces might not involve any `try/catch` statements. The fixes of system specific error handling code pieces need to adapt specific system implementations for error handling.

7/11 (63.6%) of the issues were related to errors in the form of exceptions, and their fixes were closely related to `try/catch` statements such as adding try blocks, adding catch blocks, and improving catch blocks logic. For example, in FLINK-10774 [74], Flink used a dedicated thread to retrieve topic partition information from Kafka. The thread was started whenever a Flink data consumer instance connected to Kafka for the first time, and it would be closed before Flink started fetching data from Kafka. However, if the first-time connection failed with exceptions, the fetching would not start, therefore, the thread would not be properly closed. The issue was fixed by closing the thread in a `finally` clause such that it can be closed properly even if the first-time connection fails (Figure 7.9b). We only observed one bug caused by an empty catch block in HBASE-1520 [27].

```
private volatile boolean
    isSparkConfigUpdated = false;
public HiveConf(HiveConf other) {
+   isSparkConfigUpdated =
+     other.isSparkConfigUpdated;
    ...
}
```

(a) HIVE-11250 [73]

```
try{
  ...
+ } finally {
+   partitionDiscoverer.close();
+ }
- partitionDiscoverer.close();
- kafkaFetcher.runFetchLoop();
```

(b) HBASE-1520 [27]

Figure 7.9: **(a)** The flag `isSparkConfigUpdated` indicated whether Spark-related configurations were modified or not. `isSparkConfigUpdated` was a private field of `HiveConf`. Therefore, when a `HiveConf` instance was cloned or copied, `isSparkConfigUpdated` was not automatically updated. The fix propagated the value of `isSparkConfigUpdated` when an instance of `HiveConf` was cloned or copied.
(b) `partitionDiscover` was run by a dedicated thread. The fix improved the behavior by ensuring the thread of `partitionDiscover` was properly closed even when errors occurred.

4/11 (36.4%) of the issues were related to error handling in system specific manners. For example, FLINK-17351 [51] describes a Flink failure caused by an unexpected Kafka failure as discussed in Section 5.2. When Flink detected the Kafka failure during checkpointing, the failure triggers checkpoint expiration. However, Flink would ignore the failure because the handling of `CHECKPOINT_EXPIRED` was a simple `break` as shown in Figure 7.10. The fix improved the handling of `CHECKPOINT_EXPIRED` exception type by increasing failure counters, which prevented the same exception from being ignored. In this example, errors were in the form of exceptions, but the error handling code pieces were not `try/catch` statements but function invocations and `switch` statements.

7.5 ADD FEATURE SUPPORT

10/98 (10.2%) of the issues were fixed by adding feature support. In data-plane issues, feature support mainly referred data type support (4/10). For instance, SPARK-9685 [75] shows a failure where Spark failed at querying Hive tables containing `char` typed columns because Spark did not support parsing columns with type, `char`. The fix added implementations for `char` data type.

In control-plane issues (6/10), we observed various features. For example, federation is an HDFS feature that enables scalability and data isolation. YARN-9724 [32] reports a failure caused by missing support of the federation feature in Spark. The fix added implementations for running in federation mode.

```
  public void checkFailureCounter(
    CheckpointException exception,long checkpointId) {
    ...
    CheckpointFailureReason reason = exception.getCheckpointFailureReason();
    switch (reason) {
      ...
-     case CHECKPOINT_EXPIRED:
      case JOB_FAILURE:
      case EXCEPTION:
      case TASK_FAILURE:
      case TASK_CHECKPOINT_FAILURE:
        break;
      case CHECKPOINT_DECLINED:
+     case CHECKPOINT_EXPIRED:
        if (countedCheckpointIds.add(checkpointId)) {
          continuousFailureCounter.incrementAndGet();
        }
        break;
      ...
    }
  }
```

Figure 7.10: Fix of FLINK-17351 [51] showed that error handling code pieces might not be in `try/catch` form but might utilize `switch` statements over different failure reasons that were specific to Flink implementations.

7.6 CORRECT CONFIGURATIONS

9/98 (9.2%) of the issues were fixed by correcting the passed configurations. These issues were mostly control-plane issues. (Note that we did not observe such bug fixes in the data plane, this does not mean that such bug fixes cannot take place in the data plane). When an upstream system interacts with a downstream system, the upstream system may modify behavior of the downstream system by changing the values of configuration parameters. However, the provided configuration values could be wrong, which may result in unexpected failures. FLINK-17788 [76] reports a failure where Flink started a YARN cluster whenever a Flink job was submitted. This prevented further job submissions because a YARN cluster already existed. This took place because Flink did not set the correct configuration parameter after initializing a YARN cluster the first time. The fix set the configuration which stopped double initialization of the YARN cluster (Figure 7.11).

7.7 OTHERS

A few issues were fixed by resolving race conditions on data structures, such as adding the `synchronize` keyword and replacing `LinkedList` with `CopyOnWriteArrayList`. We also

```
private def deployNewYarnCluster(config: Config, flinkConfig: Configuration) = {
  val args = parseArgList(config, "yarn-cluster")
  val commandLine = CliFrontendParser.parse(commandLineOptions, args, true)
  val executorConfig = customCLI.applyCommandLineOptionsToConfiguration(commandLine)
  val clusterDescriptor = clientFactory.createClusterDescriptor(executorConfig)
  val clusterSpecification = clientFactory.getClusterSpecification(executorConfig)
  val clusterClient = try {
    clusterDescriptor
      .deploySessionCluster(clusterSpecification)
      .getClusterClient
  } finally {
+   executorConfig.set(DeploymentOptions.TARGET, "yarn-session")
    clusterDescriptor.close()
  }
  ...
}
```

Figure 7.11: FLINK-17788 [76] was fixed by setting the correct `executorConfig` in the finally clause such that further job submissions would be deployed in `yarn-session` mode, which did not re-deploy the YARN cluster.

observed a case fixed by switching the used system client library to avoid buggy third-party implementations of the client library.

7.8 NO FIX

A few issues were not fixed but resolved by adding a more clear error message indicating that users misused the feature. For example, in SPARK-3685 [55], Spark failed at parsing a URL with an HDFS scheme for Spark parameter `spark.local.dir`, which was used to save temporary files. Instead of changing the implementation of URL parsing, developers added log messages indicating that such a parameter was expecting a local directory path. Developers insisted the design that `spark.local.dir` needed to be a local directory on the local disk and an HDFS URL should not be allowed.

CHAPTER 8: RELATED WORK

One of the main approaches to understand and characterize system reliability issues is empirical study. It provides guidance and motivations towards improving system reliability effectively and practically.

Different types of faults and errors are widely studied by reliability studies such as network issues [77, 78, 79, 80, 81], hardware faults [82, 83, 84, 85, 86], misconfigurations and software bugs [87, 88, 89, 90, 91, 92, 93], and human mistakes [94, 95, 96] in computer systems. These studies founded and inspired development and evaluation of reliability techniques.

Failure study is one type of reliability study. Failure studies explore the reasons of computer system failures, and search for techniques that overcome the failures. Starting from Jim Gray's landmark study on failures of Tandem systems [97], a number of impactful failure studies have been conducted for different types of computer systems, such as operating systems [91], storage systems [98, 99, 100], HPC systems [101, 102, 103], data-processing systems [43, 104], as well as cloud systems [44, 52, 105, 106].

Systems studied by work discussed above are mainly monolithic. However, today's distributed system infrastructures are almost never monolithically implemented. In depth study of failures for system interactions are omitted by prior studies discussed above. To our knowledge, only a few prior cloud failure studies [87] and news reports [17, 18] mentioned cross-system failures. In the large study of cloud systems [87], the authors touch on cross-system bugs related to quality-of-service, and observe that cross-system bugs are prevalent in the systems they studied, but they do not give much insight in this direction.

A recent study on production incidents on Azure [52] showed that failures in the data plane are prominent. Our study also agrees such finding that data-plane failures are prominent in cross-system failures. Moreover, we observe more diverse patterns than what was reported in [52]. For example, we find that schema mismatches and discrepancies in data type support caused considerable number of failures. Metadata for streaming such as data stream offset and data source partition information also contribute to cross-system failures. Such data-related failures were uncommon in individual systems.

CHAPTER 9: CONCLUSION

To conclude, this thesis presents a pilot study of cross-system failures in seven common cloud systems: HDFS, HBase, Flink, Spark, YARN, Hive, and Kafka. We believe that the reliability of system interactions is an important aspect to ensure reliability of distributed system infrastructures. Due to lack of work in the literature related to reliability of cross-system interactions, we were not able to find or develop effective and practical reliability techniques that improve reliability of cross-system interactions. Therefore, we conducted the pilot study in order to have a better understanding of general characteristics of cross-system failures. We studied the interaction interfaces among the seven selected systems, and studied 98 cross-system failure issues in four different dimensions: failure impacts, failure triggering, root causes, and fixes of the root causes.

- We find that the impact of cross-system failures were usually isolated at the task level. However, a considerable number of failures might impact the systems globally.
- We find that triggering events of cross-system failures can be complex. A number of logical nodes (usually more than two) were required to trigger cross-system failures. Good news is that most cross-system failures could be deterministically reproduced.
- We find that the majority (55/98, 56.1%) of cross-system failures were caused by bugs in the control plane. Bugs in the data plane also caused a significant number of cross-system failures (43/98, 43.9%).
- We find that most of the data-plane bugs (35/43, 81.4%) were fixed by adding or improving checks on shared metadata or content, and maintaining a consistent interpretation of shared data.

In the next step, with the knowledge of characteristics of cross-system failure impacts, triggering, root causes, and fixes, it is important to research the potentials of existing techniques that could help monitor, bug detect, and diagnose cross-system failures.

We believe that system interaction is an essential aspect for cloud system reliability. We hope that this thesis could serve as a first attempt to understand the characteristics of cross-system failures. Understanding of cross-system failures can shed light on designing tools and testing techniques that avoid or detect bugs that would cause cross-system failures.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotOS’10)*, 2010.

[2] “Apache Spark,” https://spark.apache.org/documentation.html.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, “The stratosphere platform for big data analytics,” *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[4] “Apache Flink,” https://flink.apache.org/.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST’10)*, May 2010.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP’03)*, Oct. 2003.

[7] “Hadoop Distributed File System,” http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[8] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC’14)*, Nov. 2014.

[9] “Alluxio,” https://docs.alluxio.io/os/user/stable/en/Overview.html.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache Hadoop YARN: Yet Another Resource Negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC’13)*, Oct. 2013.

[11] “Apache Hadoop YARN,” https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI’11)*, Mar. 2011.

[13] “Apache Mesos,” http://mesos.apache.org/documentation/latest/.

[14] “etcd,” https://etcd.io/docs/.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*, June 2010.

[16] "Apache ZooKeeper," https://zookeeper.apache.org/.

[17] A. Hern, "Google suffers global outage with gmail, youtube and majority of services affected," https://www.theguardian.com/technology/2020/dec/14/google-suffers-worldwide-outage-with-gmail-youtube-and-other-services-down, Dec. 2020.

[18] J. Kastrenakes, "Facebook outage disrupted messenger and instagram dms," https://www.theverge.com/2020/12/10/22167328/facebook-messenger-down-instagram-dms-outage, Dec. 2020.

[19] SPARK-27239, "Processing compressed hdfs files with spark failing with error: "java.lang.illegalargumentexception: requirement failed: length (-1) cannot be negative" from spark 2.2.x," https://issues.apache.org/jira/browse/SPARK-27239, 2019.

[20] FLINK-12342, "Yarn resource manager acquires too many containers," https://issues.apache.org/jira/browse/FLINK-12342, 2019.

[21] C. Zamfir, G. Altekar, and I. Stoica, "Automating the debugging of datacenter applications with adda," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[22] "Apache Hive," https://cwiki.apache.org/confluence/display/Hive/Tutorial.

[23] "Apache Kafka," https://kafka.apache.org/documentation/.

[24] "Apache HBase," https://hbase.apache.org/book.html.

[25] "ASF JIRA," https://issues.apache.org/jira/secure/Dashboard.jspa.

[26] YARN-2790, "Nm can't aggregate logs past hdfs delegation token expiry." https://issues.apache.org/jira/browse/YARN-2790, 2014.

[27] HBASE-1520, "Storefilescanner catches and ignore ioexceptions from hfile," https://issues.apache.org/jira/browse/HBASE-1520, 2009.

[28] SPARK-25206, "wrong records are returned when hive metastore schema and parquet schema are in different letter cases," https://issues.apache.org/jira/browse/SPARK-25206, 2018.

[29] HIVE-3355, "Special characters (such as '') displayed as '?' in hive," https://issues.apache.org/jira/browse/HIVE-3355, 2012.

[30] YARN-8223, "Classnotfoundexception when auxiliary service is loaded from hdfs," https://issues.apache.org/jira/browse/YARN-8223, 2018.

[31] SPARK-15046, "When running hive-thriftserver with yarn on a secure cluster the workers fail with java.lang.numberformatexception," https://issues.apache.org/jira/browse/SPARK-15046, 2016.

[32] YARN-9724, "Error sparkcontext: Error initializing sparkcontext." https://issues.apache.org/jira/browse/YARN-9724, 2019.

[33] FLINK-3067, "Kafka source fails during checkpoint notifications with npe," https://issues.apache.org/jira/browse/FLINK-3067, 2015.

[34] SPARK-5164, "Yarn — spark job submits from windows machine to a linux yarn cluster fail," https://issues.apache.org/jira/browse/SPARK-5164, 2015.

[35] SPARK-18968, ".sparkstaging quickly fill up hdfs," https://issues.apache.org/jira/browse/SPARK-18968, 2016.

[36] FLINK-8638, "Job restart when checkpoint on barrier failed," https://issues.apache.org/jira/browse/FLINK-8638, 2018.

[37] FLINK-4486, "Jobmanager not fully running when yarn-session.sh finishes," https://issues.apache.org/jira/browse/FLINK-4486, 2016.

[38] HIVE-11166, "Hivehbasetableoutputformat can't call getfileextension(jobconf jc, boolean iscompressed, hiveoutputformat ?, ? hiveoutputformat)," https://issues.apache.org/jira/browse/HIVE-11166, 2015.

[39] SPARK-19361, "kafka.maxrateperpartition for compacted topic cause exception," https://issues.apache.org/jira/browse/SPARK-19361, 2017.

[40] SPARK-1111, "Url validation throws error for hdfs url's," https://issues.apache.org/jira/browse/SPARK-1111, 2014.

[41] SPARK-8374, "Job frequently hangs after yarn preemption," https://issues.apache.org/jira/browse/SPARK-8374, 2015.

[42] FLINK-3440, "Kafka should also checkpoint partitions where no initial offset was retrieved," https://issues.apache.org/jira/browse/FLINK-3440, 2016.

[43] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[44] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, 2016.

[45] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.

[46] SPARK-18220, "Classcastexception occurs when using select query on orc file," https://issues.apache.org/jira/browse/SPARK-18220, 2016.

[47] FLINK-13197, "Fix hive view row type mismatch when expanding in planner," https://issues.apache.org/jira/browse/FLINK-13197, 2019.

[48] HBASE-16621, "Hbck should have -fixhfilelinks," https://issues.apache.org/jira/browse/HBASE-16621, 2017.

[49] FLINK-17189, "Table with processing time attribute can not be read from hive catalog," https://issues.apache.org/jira/browse/FLINK-17189, 2020.

[50] FLINK-7143, "Partition assignment for kafka consumer is not stable," https://issues.apache.org/jira/browse/FLINK-7143, 2017.

[51] FLINK-17351, "Checkpointcoordinator and checkpointfailuremanager ignores checkpoint timeouts," https://issues.apache.org/jira/browse/FLINK-17351, 2020.

[52] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What Bugs Cause Production Cloud Incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*, May 2019.

[53] SPARK-16605, "Spark2.0 cannot "select" data from a table stored as an orc file which has been created by hive while hive or spark1.6 supports," https://issues.apache.org/jira/browse/SPARK-16605, 2016.

[54] HIVE-17002, "decimal (binary) is not working when creating external table for hbase," https://issues.apache.org/jira/browse/HIVE-17002, 2017.

[55] SPARK-3685, "Spark's local dir should accept only local paths," https://issues.apache.org/jira/browse/SPARK-3685, 2014.

[56] SPARK-4267, "Failing to launch jobs on spark on yarn with hadoop 2.5.0 or later," https://issues.apache.org/jira/browse/SPARK-4267, 2014.

[57] SPARK-17000, "Spark cannot connect to secure metastore when using custom metastore jars," https://issues.apache.org/jira/browse/SPARK-17000, 2016.

[58] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*, Nov. 2013.

[59] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early Detection of Configuration Errors to Reduce Failure Damage," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Nov. 2016.

[60] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An Evolutionary Study of Configuration Design and Implementation in Cloud Systems," in *In Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*, May 2021.

[61] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[62] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*, July 2021.

[63] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Nov. 2020.

[64] X. Sun, L. Suresh, A. Ganesan, R. Alagappan, M. Gasch, L. Tang, and T. Xu, "Reasoning about modern datacenter infrastructures using partial histories," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, May 2021.

[65] FLINK-9349, "Kafkaconnector exception while fetching from multiple kafka topics," https://issues.apache.org/jira/browse/FLINK-9349, 2018.

[66] HIVE-19266, "Use udfs in hive-on-spark complains unable to find class exception regarding kryo," https://issues.apache.org/jira/browse/HIVE-19266, 2018.

[67] FLINK-3156, "Flinkkafkaconsumer fails with npe on notifycheckpointcomplete," https://issues.apache.org/jira/browse/FLINK-3156, 2015.

[68] FLINK-15194, "Directories in distributed caches are not extracted in yarn per job cluster mode," https://issues.apache.org/jira/browse/FLINK-15194, 2019.

[69] FLINK-15429, "Hiveobjectconversion implementations need to handle null values," https://issues.apache.org/jira/browse/FLINK-15429, 2019.

[70] HBASE-10029, "Proxy created by hfilesystem#createreorderingproxy() should properly close when connecting to ha namenode," https://issues.apache.org/jira/browse/HBASE-10029, 2013.

[71] SPARK-21686, "spark.sql.hive.convertmetastoreorc is causing nullpointerexception while reading orc tables)," https://issues.apache.org/jira/browse/SPARK-21686, 2017.

[72] FLINK-9384, "Kafkaavrotablesource failed to work due to type mismatch," https://issues.apache.org/jira/browse/FLINK-9384, 2018.

[73] HIVE-11250, "Change in spark.executor.instances (and others) doesn't take effect after rsc is launched for hs2 [spark brnach]," https://issues.apache.org/jira/browse/HIVE-11250, 2015.

[74] FLINK-10774, "connection leak when partition discovery is disabled and open throws exception," https://issues.apache.org/jira/browse/FLINK-10774, 2018.

[75] SPARK-9685, ""unsupported datatype: char(x)" in hive," https://issues.apache.org/jira/browse/SPARK-9685, 2015.

[76] FLINK-17788, "scala shell in yarn mode is broken," https://issues.apache.org/jira/browse/FLINK-17788, 2020.

[77] J. Meza, T. Xu, K. Veeraraghavan, and Y. J. Song, "A Large Scale Study of Data Center Network Reliability," in *Proceedings of the 2018 ACM Internet Measurement Conference (IMC'18)*, Oct. 2018.

[78] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Aug. 2016.

[79] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," in *Proceedings of the 2011 ACM SIGCOMM Conference (SIGCOMM'11)*, Oct. 2011.

[80] R. Potharaju and N. Jain, "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters," in *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC'13)*, Oct. 2013.

[81] Y. Li, H. Lin, Z. Li, L. Gong, F. Qian, Y. Liu, X. Xin, and T. Xu, "A Nationwide Study on Cellular Reliability: Measurement, Analysis, and Enhancements," in *Proceedings of the 2020 Annual Conference of the ACM Special Interest Group on Data Communication (Sigcomm'21)*, Aug. 2021.

[82] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Feb. 2018.

[83] B. Schroeder and G. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, Feb. 2007.

[84] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, Feb. 2009.

[85] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A Large-scale Study of Flash Memory Errors in the Field," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15)*, June 2015.

[86] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Characteristics of DRAM Errors and the Implications for System Design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, Mar. 2012.

[87] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*, Nov. 2014.

[88] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'11)*, Sep. 2011.

[89] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A Study of Linux File System Evolution," *ACM Trans. Storage*, vol. 10, no. 1, Jan. 2014.

[90] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 329–339, Mar. 2008.

[91] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 73–88, Oct. 2001.

[92] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, July 2015.

[93] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Aug. 2015.

[94] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How Do System Administrators Resolve Access-Denied Issues in the Real World?" in *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*, May 2017.

[95] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and Dealing with Operator Mistakes in Internet Services," in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[96] T. Xu, V. Pandey, and S. Klemmer, "An HCI View of Configuration Problems," *arXiv:1601.01747*, Jan. 2016.

[97] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Tandem Technical Report 85.7*, June 1985.

[98] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, Feb. 2008.

[99] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems," in *Proceedings of the 9th USENIX Conference on Operating SystemsDesign and Implementation (OSDI'10)*, Oct. 2010.

[100] G. Amvrosiadis and M. Bhadkamkar, "Getting Back Up: Understanding How Enterprise Data Backups Fail," in *Proceedings of 2016 USENIX Annual Technical Conference (ATC'16)*, June 2016.

[101] S. Kendrick, "What Takes Us Down?" *USENIX ;login:*, vol. 37, no. 5, pp. 37–45, Oct. 2012.

[102] C. D. Martino, Z. T. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, June 2014.

[103] S. Jha, S. Cui, S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live Forensics for HPC Systems: A Case Study on Distributed Storage Systems," in *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC'20)*, Nov. 2020.

[104] A. Rabkin and R. Katz, "How Hadoop Clusters Break," *IEEE Software Magazine*, vol. 30, no. 4, pp. 88–94, July 2013.

[105] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," in *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, Oct. 2016.

[106] K. Veeraraghavan, J. Meza, S. Michelson, S. Panneerselvam, A. Gyori, D. Chou, S. Margulis, D. Obenshain, S. Padmanabha, A. Shah, Y. J. Song, and T. Xu, "Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Oct. 2018.