Vicious Cycles in Distributed Software Systems

Shangshu Qian
Purdue University
West Lafayette, USA
shangshu@purdue.edu

Wen Fan
Purdue University
West Lafayette, USA
fan372@purdue.edu

Lin Tan
Purdue University
West Lafayette, USA
lintan@purdue.edu

Yongle Zhang
Purdue University
West Lafayette, USA
yonglezh@purdue.edu

***Abstract*—A major threat to distributed software systems' reliability is vicious cycles, which are observed when an event in the distributed software system's execution causes a system degradation, and the degradation, in turn, causes more of such events. Vicious cycles often result in large-scale cloud outages that are hard to recover from due to their self-reinforcing nature.**

This paper formally defines Vicious Cycle, and conducts the first in-depth study of 33 real-world vicious cycles in 13 widely-used *open-source* distributed software systems, shedding light on the root causes, triggering conditions, and fixing strategies of vicious cycles, with over a dozen concrete implications to combat them. Our findings show that the majority of the vicious cycles are caused by incorrect error handlers, where the handlers do not obtain enough information to distinguish between 1) an error induced by incoming requests and 2) an error induced by an unexpected interference from another error handler.

This paper further performs a feasibility study by 1) building a monitoring tool that prevents one type of vicious cycle by collecting information to make a more informed decision in error handling, and 2) investigating the effectiveness of one commonly suggested practice—injecting exponential backoff—to prevent vicious cycles induced by unconstrained retry.

***Index Terms*—distributed software systems, vicious cycles**

I. Introduction

Internet services today live on data-intensive distributed software systems such as distributed storage systems and distributed computation frameworks. Such distributed software systems are designed to be highly reliable by tolerating component failures with technologies such as data replication [17], [31], [46] , and recomputation [96]. Unfortunately, cascading component failures still happen and cause severe consequences such as service outages. Such cascading failures happen because software defects and design flaws propagate failures from one component to another. Cascading failures often manifest through a *Vicious Cycle* (VC), in which an event in the distributed software system's execution causes a system degradation, and the degradation, in turn, causes more of such events.

An incident [41] from Amazon Web Services (AWS) shows how a temporary network failure disrupts the entire Amazon Elastic Block Store (EBS) service through a vicious cycle: Data stored on EBS are replicated on multiple storage servers. When a temporary network failure causes some storage servers to lose connection to their mirrors, they start replicating data to other storage servers. Unfortunately, these replication requests trigger a latent race condition that causes more storage servers to crash, resulting in more replication requests. This forms a self-reinforcing cycle that brought down the entire EBS service and was only stopped by manually disabling EBS replication requests and adding capacities to the cluster.

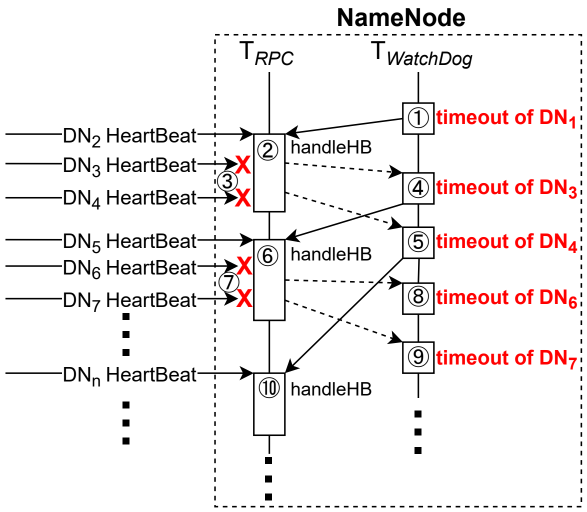


Fig. 1: Hadoop-572: By clogging up the NameNode's RPC thread (T_{RPC}), DataNode DN_1's timeout causes other DataNodes (i.e., DN_3 and DN_4) to time out, which further causes additional DataNodes (i.e., DN_6 and DN_7) to time out, which eventually causes the entire cluster to crash. DN is DataNode. HB denotes HeartBeat.

Vicious Cycles are prevalent in commercial distributed software systems. At the time of writing, a third [40]–[44] of the 15 severe incidents reported by AWS [13] in the past 12 years manifest as vicious cycles. In fact, some distributed software system practitioner claims that once a system reaches a certain level of reliability, most major incidents will involve a vicious cycle [15]. Thus, there is high demand for a deep understanding of such vicious cycles so that one can build avoidance, testing, diagnosis, and recovery techniques for them. Unfortunately, postmortem reports from cloud vendors typically provide limited information. To fill this gap, we conduct the first in-depth study of 33 vicious cycles on 13 **open-source** distributed software systems. Our source-code level analysis reveals unique challenges in combating vicious cycles, as illustrated below.

A. A Motivating Example

Figure 1 shows Hadoop-572 [26], a real-world vicious cycle in Hadoop Distributed File System (HDFS), which has a similar characteristic as the AWS incident discussed above. The failure happens in a large Hadoop cluster of 600 *DataNodes*, where one DataNode holds over 10,000 data blocks, and a *NameNode* stores the metadata. A NameNode has a watchdog thread $T_{WatchDog}$ that periodically checks the timestamp of the last heartbeat from each DataNode and a remote procedure call (RPC) thread T_{RPC} that receives and processes HeartBeat messages from DataNodes.

The vicious cycle starts when NameNode misses DN_1's HeartBeat message. Its $T_{WatchDog}$ detects DataNode DN_1's timeout failure and marks blocks stored on DN_1 as under-replicated (① in Figure 1). When the NameNode receives a heartbeat from another DataNode DN_2, the NameNode generates replication requests for these under-replicated blocks that were stored on DN_1, and wraps them in the heartbeat response to DN_2 (②). Unfortunately, generating replication requests occupies the NameNode's RPC thread (T_{RPC}) and prevents the delivery of heartbeats from other DataNodes (i.e., DN_3 and DN_4) (③) to T_{RPC}. Although the heartbeat timeout threshold is configured as one minute to allow retries, a software defect limits the number of heartbeat retries and renders the one-minute timeout threshold ineffective. The watchdog thread $T_{WatchDog}$ mistakenly marks the affected DataNodes (i.e., DN_3 and DN_4) as dead (④ and ⑤) and marks more blocks on DN_3 and DN_4 as under-replicated, causing more workload for the NameNode itself (⑥ and ⑩).

Upon receiving a heartbeat message from DN_5, the NameNode generates and sends replication requests to DN_5 for data blocks on DN_3 (⑥) and prevents the delivery of the heartbeat from other DataNodes (i.e., DN_6 and DN_7) through the same process described above (⑦), again causing more workload on the NameNode (⑧ and ⑨). The entire Hadoop cluster then falls into a vicious cycle and crashes.

The root cause of Hadoop-572 lies in the *error handler* of DataNode failures (i.e., $T_{Watchdog}$), whose *recovery task* causes undesired *interference* that results in a HeartBeat miss for another DataNode. The *recovery task* performed is generating the replication requests, which *interferes* with the delivery of the HeartBeat messages. The error handler does not have enough information to distinguish between a DataNode HeartBeat miss caused by 1) a real DataNode crash and 2) a delayed HeartBeat delivery due to interference from a previous recovery task. The confusion of the error handler makes the system susceptible to vicious cycles.

To prevent this vicious cycle, developers apply a series of fixes. They 1) fix the software defect that limits DataNode heartbeat retries, 2) optimize the NameNode's performance, and 3) avoid unnecessary replication requests when a dead DataNode is resurrected. Applying any of the fixes alone cannot avoid the vicious cycle, demonstrating the complexity of fixing vicious cycles (details below in "*Hard to repair*").

This example shows that vicious cycles are particularly difficult to tackle because of the following challenges:

1) *Complex triggering conditions.* Vicious cycles typically require a unique combination of workload, cluster configuration, and fault injection in cluster-level testing (e.g., integration testing [75]) to be triggered. For example, the prolonged heartbeat handling in Figure 1 requires each failed DataNode to hold over 10,000 data blocks. Unfortunately, thoroughly exploring the input space, configuration space, and fault injection space in cluster-level testing is extremely challenging [61], which makes vicious cycles difficult to be exposed before software release.
2) *Hard to recover.* A vicious cycle involves events and system degradations that aggravate each other. Oftentimes, it is the automated failure recovery stage itself aggravating the degradation. In addition, this process could quickly develop into a cluster outage due to amplification and leave little chance for manual intervention and recovery. For example, the vicious cycle in Figure 1 could result in a cluster outage within a few heartbeat timeout intervals.
3) *Hard to repair.* A vicious cycle typically happens due to both design flaws (e.g., poor fault isolation and performance bug) and logic errors. Thus, fixing one of them often fails to prevent similar failures in the future. For the vicious cycle in Figure 1, although fixing the software defect alone would allow the DataNode to retry the heartbeat within the one-minute timeout interval, the generation of the replication request could exceed this limit given enough under-replicated data blocks, causing the entire cluster to crash. With either of the latter two fixes alone (fixes 2) or 3)), the DataNode's heartbeat would still be highly likely to time out, because the number of retries is incorrectly limited, reducing the probability of heartbeat delivery.

B. Contribution

This paper defines the concept of the vicious cycle (§II) and provides the first comprehensive, in-depth analysis of the vicious cycle in real-world distributed software systems. Specifically, we study 33 real-world vicious cycles from 13 widely-used, *open-source* distributed software systems, namely, Apache Cassandra [3], HDFS [34], HBase [6], ZooKeeper [12], Hadoop [5], Kafka [8], Flink [4], Solr [10], Ignite [1], ActiveMQ [2], Storm [11], Accumulo [1], and Ratis [9]. For each case, we carefully analyzed its report and source code to thoroughly understand the root cause of the vicious cycle, the triggering condition, and the fix strategy. In addition, we reproduced eight of the cases to better understand them. This paper makes the following contributions:

- **A symptom study** (§IV). We show that vicious cycles have severe consequences.
  - Most (82%) vicious cycles result in node failures (58%) and elevated queue size (24%).
  - A majority (55%) of vicious cycles **amplify** the error, causing it to grow exponentially.
- **A root-cause study** (§V). We find that vicious cycles are formed either 1) (Unexpected Error) when an undetected or unhandled error propagates along a global cycle (40%) or 2) (Unexpected Cycle) when error handlers unexpectedly interfere with request handling and cause other requests to fail (60%).

  - Unexpected Error: 18% of the vicious cycles are caused by undetected errors, which are highly diverse and system-specific. Another 22% are caused by unhandled errors, all happen when retrying an error-inducing input.
  - Unexpected Cycle: More than a third (36%) of vicious cycles happen due to incorrect decisions when recovering from system degradation. The remaining 24% of vicious cycles are formed due to unconstrained retries.
- **A triggering-condition study** (§VI). We find that vicious cycles have unique requirements to be triggered.
  - Many (42%) vicious cycles require a heavy workload.
  - A large portion (36%) of vicious cycles are non-deterministic, and a non-negligible amount (18%) requires timing constraints with very small time windows.
  - 24% of vicious cycles can only be triggered when the cluster is in a special state, such as during a rolling upgrade or when nodes are holding large amounts of data.
- **A fixing-strategy study** (§VII). We find that vicious cycles are difficult to fix.
  - A significant portion (61%) of the vicious cycles are fixed by major redesigns, including 1) reducing the workload (18%), 2) separating heavy workload (6%), and 3) system-specific redesigns (37%).
  - A surprising amount (27%) of fixes are considered workarounds by developers but not complete fixes.
- A feasibility study (§VIII) to prevent vicious cycles through well-informed error handling and exponential backoff.
- The **first data set** [45] of 33 real-world vicious cycle bugs, from 13,455 bug reports, in 13 **open-source** projects.
- Actionable **implications and guidelines** to prevent vicious cycles. These include suggestions for runtime detection and prevention techniques, testing techniques, and good practices in software development to avoid vicious cycles.

II. Definitions

In this section, we give a concrete definition of the vicious cycle, as, to the best of our knowledge, such a definition does not exist in any previous works. This definition is necessary to identify vicious cycles and is applied to each studied case. An example of applying our definition to Hadoop-572 (§I) is presented later in this section.

Definition 1 *A **vicious cycle** is a **global iterative execution** in which each **iteration** results in a **system degradation** and the degradation in turn causes one or multiple future iterations.*

When one iteration in a vicious cycle causes *multiple* (as opposed to one) future iterations (through the degradation), the vicious cycle has an **amplification behavior**. A vicious cycle with an amplification behavior typically has the most severe consequences.

A **global iterative execution** is defined using a **causality graph** that represents the execution of the entire distributed software system and captures the causal relationships between system degradations and iterative executions. We first the define causality graph as follows:

Definition 2 *A causality graph, $G(V, E)$, is a directed acyclic graph with each vertex $v \in V$ representing an event that happened during the execution and each edge $e \in E$ representing a causal relationship between a pair of events. Each vertex is labeled with the corresponding event, and each edge is labeled with the corresponding causal relationship.*

Events and causal relationships are defined with enumerative definitions in §II-A. One example of an event is receiving a network message. One example of the causal relationship is the *happens-before* relationship [72] between a pair of sending and receiving a network message.

With the definition of causality graph, we define **global iterative execution** as follows:

Definition 3 *An **iterative execution** is formed in a causality graph when there are multiple isomorphic subgraphs connected through edges. Each subgraph is called an iteration. The isomorphism comparison considers subgraphs' structure, as well as their vertices' and edges' labels. A **global iterative execution** is an iterative execution in which events of each isomorphic subgraph are spread across multiple threads or nodes.*

The notion of **system degradation** involved in a vicious cycle could range from general notions such as an increasing number of crashed nodes to system-specific notions such as an increasing number of missing file blocks in a distributed file system. We define a system degradation using a *set point* and a *measurement function*, both of which can be system-specific.

Definition 4 *A set point (n_{sp}) is a numeric value corresponding to a property that the system should satisfy in an ideal state. This property is in the form of a predicate over a measurable system state (n_{st}) and the set point: $n_{st} == n_{sp}$. A measurement function ($f_m()$) measures the distance between the current system state and the set point: $f_m() = n_{st} - n_{sp}$. A **system degradation** happens when the result of executing $f_m()$ over the current state is larger than that of a previous state: $f_m() - f'_m() > 0$.*

In Hadoop-572 [26] (§I), the property that the system should satisfy in an ideal state is $crashed\_nodes.size() == 0$ (i.e., $n_{st} = crashed\_nodes.size()$), and the *set point* $n_{sp} == 0$. The *measurement function* is $f_m() = crashed\_nodes.size() - 0$.

A. Events and Causal Relationships

Events. Events in a causality graph belong to the following two categories:

1) Errors event that results in system degradation. Such events could be system-specific and should be handled properly by an **error handler**.
2) Normal execution logic such as request handling, RPC call procedures, etc.

Causal relationships. Causal relationships in a distributed software system refer to the *happens-before* relationship [72] and its variants [77], [78], [84]. Specifically, the causal relationships include the following four categories:

1) *Inter-node network communication* Sending a network message through sockets happens before its reception.
2) *Intra-node multi-thread communication* Submitting a task to a thread pool happens before its execution, and its execution happens before its join. Similarly, multi-thread communication includes the happens-before relationship between thread creation, thread execution, and thread join.
3) *Intra-thread happens-before* An event happens before another event that occurs later in the same thread. When the thread is an event handler thread, this relationship only applies to events in the same invocation of the handler.
4) *Resource contention* A task submission to a thread pool has a potential resource contention with later task submissions.

B. Identifying Vicious Cycles

To determine whether a studied case involves a vicious cycle, we construct its causality graph and analyze whether its causality graph contains a global iterative execution with a system degradation. We illustrate this process with the real-world vicious cycle in our introduction (§I): Hadoop-572 [26].

Figure 1 shows a simplified causality graph we constructed for Hadoop-572. Events are presented with boxes (e.g., the timeout error event of DataNodes) labeled with the event name. Some event boxes (e.g., the heartbeat from DataNodes) are omitted due to space limitations. Causal relationships are presented with lines and arrows. Solid lines represent happens-before relationships. Solid arrows are constructed from inter-node (e.g., DataNode heartbeat triggering RPC handler) and inter-thread communication (e.g., timeout error of a DataNode cause RPC handler to replicate the DataNode's blocks). Red cross with dashed arrows represent resource contention (e.g., heartbeat from a DataNode is not delivered due to contention and causing its timeout error).

With this causality graph, we identify two isomorphic subgraphs: [①, ②, ③] and [④, ⑥, ⑦], and the execution in each subgraph involves the NameNode and multiple DataNodes. Therefore, Hadoop-572 is a *global iterative execution* (*Definition 3*), and each subgraph makes up of one *iteration*. Note that three more incomplete iterations exist in Figure 1: [⑤, ⑩], [⑧], and [⑨].

The system degradation (*Definition 4*) is quantified by the number of crashed nodes in the cluster, with the set point being 0. Within each iteration, one more DataNode is marked as dead by the NameNode. Therefore, each global iteration results in a system degradation (*Definition 1*), and Hadoop-572 is a vicious cycle by our definition.

III. Experimental Methods

We conducted the study on 33 vicious cycles from a wide range of popular open-source distributed software systems, as shown in Table I, including distributed databases [3], [6], [7], key-value stores [1], [12], filesystem [34], streaming processing systems [2], [4], [8], [11], computing framework [5], consensus library [9], and search platform [10].

We selected the set of failures from the issue trackers of the distributed software systems above. We search for resolved and valid issues whose bug report contains a list of keywords, including "vicious cycle, vicious circle, cascade, spiral, feedback loop, multiplying effect, amplify, overwhelm, storm, bounce, snowball, chain reaction, and domino".

TABLE I: The number of vicious cycles in each system.

System	#	System	#	System	#
HBase [6]	6	Solr [10]	3	Accumulo [1]	1
HDFS [34]	6	Flink [4]	2	Ignite [7]	1
Hadoop [5]	4	Storm [11]	1	ZooKeeper [12]	1
Kafka [8]	3	Ratis [9]	1	ActiveMQ [2]	1
Cassandra [3]	3			**Total**	**33**

Our keywords are selected using a multi-round boosting strategy combined with manual review: We start with a small set of reasonable keywords and collect all the bug candidates (*filtering step*). Then, we manually go through the bug candidates collected by the filtering step, excluding cases that are not a vicious cycle, and identify new keywords based on the title, description, and discussion associated with each candidate (*reviewing step*). With the new set of keywords, we start a new round of the filtering process. With such boosting strategy, we are able to expand the number of bugs in each round and keep the representativeness of the keywords. This process stops when we could not identify new keywords. Our manual filtering exhausted all the bug reports containing these keywords. In the end, we get 33 vicious cycles as shown in Table I from 13,455 bug reports in total.

Among the 33 cases, about one-third (30%) are reported within recent five years. A majority (91%) of them have a priority of major or higher in the bug tracking system. One bug (HBase-27149 [30]) was reported in 2022 and another bug (Kafka-10888 [35]) was not resolved until 2022, showing that vicious cycles are still present in recent systems.

Our analysis results for all 33 vicious cycles are available in an anonymous repository [45]. The threats to validity of our filtering process are discussed in §IX.

IV. Symptoms of Vicious Cycles

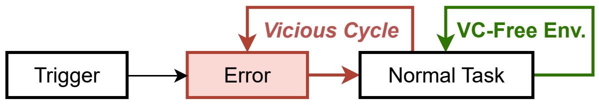
We first investigate the manifestations of vicious cycles, focusing on their symptoms and propagation.

Finding 1: Most (82%) vicious cycles result in easily observable symptoms such as node failures (58%) and elevated queue size (24%). The remaining (18%) has system-specific degradation. Majority (55%) of vicious cycles have amplification behavior.

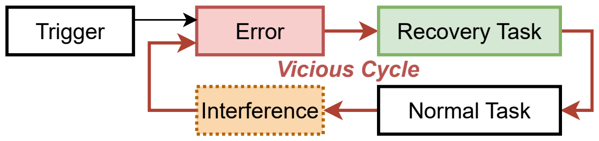
As defined in §II, one iteration of a vicious cycle with amplification behavior can cause multiple subsequent iterations. A vicious cycle with the amplification behavior is the most severe subtype as it is usually hard to recover from and can quickly propagate the system degradation to the entire cluster. For example, **Hadoop-572** [26] (§I) has an amplification behavior, where the timeout of one DataNode could result in timeouts of multiple DataNodes.

TABLE II: An overview of error handler's role in the formation of vicious cycle. Definitions of each type and subtype of vicious cycles are italicized. The number in the parenthesis indicates the percentage of vicious cycles in each category.

Vicious Cycle Type	Subtype	Interference
Unexpected Error (40%) *An error that hinders the task completion is propagated along a global cycle.*	**Undetected Error (18%)** *The error in the cycle is silent, thus no error handler is implemented.*	N/A
	Unhanded Error (22%) *The error in the cycle is observed, but not properly handled.*	N/A
Unexpected Cycle (60%) *The error handlers unexpectedly interfere with request handling and cause other requests to fail.*	**Incorrect Degradation Recovery (36%)** *The interference from the degradation recovery tasks causes further degradation.*	Performance Interference (21%)
		Functional Interference (15%)
	Unconstrained Retry (24%) *Previous retries of a request interfere with the current retry, causing the current one to fail.*	Performance Interference (24%)



(a) Conceptual graph of a UE vicious cycle. The red arrows indicate a vicious cycle. The green arrow indicate a VC-free environment.



(b) Conceptual graph of a UC vicious cycle. The red arrows indicate a vicious cycle.

Fig. 2: Conceptual graph of vicious cycles.

Implication: Most vicious cycles can grow exponentially in the cluster, propagating the system degradation rapidly across the cluster. Detection and prevention techniques need to intervene as soon as possible to prevent a potential whole-cluster collapse.

V. Root Causes of Vicious Cycles

The key component forming a vicious cycle is the error handlers. Thus, depending on error handlers' interaction with the global iterative execution or the absence of error handlers, we categorize vicious cycles based on their root causes into two types – **Unexpected Error** (UE), an error in a global cycle is undetected or unhandled, and **Unexpected Cycle** (UC), when error handlers unexpectedly interfere with request handling and cause other requests to fail.

Figure 2a and Figure 2b illustrate the UE and UC vicious cycles respectively. Figure 2a shows a UE vicious cycle (red arrows) where an **error** (red box) affects normal execution which causes more errors. The green VC-free cycle where multiple normal executions (e.g., requests) continue error-free.

Figure 2b shows a UC vicious cycle (red arrows). An error handler performs a **recovery task** (green box) to recover from the error. However, the recovery task interferes with other requests and causes them to run into the same error. The error handler lacks information to distinguish between errors caused by the **interference** (orange dotted box) and errors caused by the external trigger (more discussion in §VI), forming a vicious cycle (the cycle formed by the red arrows).

Table II shows our root cause taxonomy of two VC types (UE and UC) and four VC subtypes. Numbers in the parenthesis are the percentage of vicious cycles belonging to each category. Two major causes of UE vicious cycles are 1) undetected error (§V-A), and 2) unhandled error (§V-B). We further classify the UC vicious cycles based on the error handler's recovery task in each iteration: 1) incorrect degradation recovery (§V-C), and 2) unconstrained retry (§V-D).

Column "Interference" shows the type of interferences between recovery tasks and requests in a vicious cycle. For UE vicious cycles, errors are either silent or not properly handled. Thus, no interference originates from the error handler. For UC vicious cycles, we observe two types of interference—performance interference and functional interference. Below we explain these VC types and subtypes and their interaction with interference with examples.

Finding 2: Unexpected errors make up 40% of vicious cycles, when an undetected or unhandled error propagates along a global cycle.

An example of an undetected error is **Kafka-10888** [35], where a slow node is caused by a buggy load balancer. No explicit error such as an exception is thrown, and no error handler is implemented.

One example of an unhandled error is **HBase-14598** [28], where an HBase client retries a bad request causing a server crash. The client blindly retries the bad request because the server crash is never properly handled.

Finding 3: Majority (60%) of vicious cycles are unexpected cycles, when error handlers unexpectedly interfere with request handling and cause other requests to fail.

For example, as in **Hadoop-572** [26] (§I), the error is the delay of DataNode heartbeat delivery. The error handler, replicating the under-replicated blocks, cannot differentiate between a delay caused by an actual DataNode failure (thus no heartbeat message is sent) and a contention in the RPC handler thread pool caused by the recovery task (thus a heartbeat message is received but not delivered).

Interference. As shown in Table II, both performance and functional interference can cause vicious cycles. Performance interferences include contentions through CPU resources, memory resources, and a lock. For example, in **Hadoop-572** [26] (§I) the recovery task interferes with the heartbeat through a CPU contention (a shared thread pool).

Functional interferences include IO errors, deadlocks, and other system-specific errors caused by the recovery task and interrupt request handling. For example, in **HDFS-12914** [32], the block replication (recovery task) triggers a deadlock on the DataNode, causing its block report to be rejected by the NameNode and more block replication initiated.

Note that an unconstrained retry could cause functional interference, but we do not observe it in our study.

A. UE Subtype 1: Undetected Error

Finding 4: About one-sixth (18%) of the vicious cycles are caused by undetected errors (silent errors). **Majority** (83%) of such cases happen due to logic errors in the code.

For example, **Cassandra-13441** [14] is a vicious cycle that happens due to a silent error propagated through Cassandra's gossip protocol. Due to a silent logic error in Cassandra 3.0.14, when a node is upgraded with no scheme change, gossip messages [56], [70] between this node and other non-upgraded nodes indicate a schema mismatch by mistake. The schema mismatch incurs schema migration requests between the upgraded and non-upgraded nodes. Non-upgraded nodes which have detected the schema mismatch further spread information about the schema mismatch to the remaining nodes in the cluster through Cassandra's gossip protocol, resulting in a storm of unnecessary schema migration requests. The silent error is spread to the entire cluster through the gossip protocol, forming a vicious cycle.

We find that logic errors that cause silent errors are highly diverse and system-specific and, unfortunately, could not identify general bug patterns. As discussed above, the logic error in **Cassandra-13441** is closely coupled with the gossip protocol.

Implication: The system-specific nature of the logic errors makes vicious cycles triggered by them hard to be found during the testing phase. Fortunately, logic errors usually accompany observable system degradations (§IV), indicating opportunities for automated detection.

B. UE Subtype 2: Unhandled Error

Finding 5: Another important (22%) reason for vicious cycles is unhandled errors induced by "deadly retry", where a client retries a request that causes unhandled errors, such as a node crash, a node getting stuck, or the RPC queue being flooded, on different nodes.

For example, **HBase-14598** [28] is such a vicious cycle: The bug is caused by a problematic Scan (an HBase query) allocating a large chunk of memory, larger than that is allowed by the JVM. Such a request results in an OutOfMemoryError (OOM) and crashes the RegionServer (RS) processing the request. The problematic Scan is then retried on a new RS and causes it to fail as well.

Implication: Vicious cycles caused by deadly retries result in fatal errors that are detected but not handled. The reason is the lack of the ability to infer the causal relationship between the error-inducing retried request and the error, i.e., the system does not know whom to blame when the error happens. Once this causal relationship is identified, the error-inducing request can be blocked to break the vicious cycle. More details about detecting erroneous inputs are discussed in §VIII-A.

C. UC Subtype 1: Incorrect Decision in Degradation Recovery

Finding 6: More than a third (36%) of the vicious cycles happen due to incorrect decisions when recovering from the system degradation.

Specifically, degradation recovery includes capacity recovery (24%) such as node restart and reconnection, and data recovery (12%) such as replicating missing file blocks. As shown in Figure 2b, such an incorrect decision typically happens when the error handler is not able to distinguish an error caused by an external trigger and an error caused by the **interference** of the error handling execution itself.

Finding 7: Majority of the interferences caused by degradation recovery are contention in CPU (42%) or memory (17%) resources, resulting in heartbeat delays, prolonged garbage collection (GC) periods, or out-of-memory errors. The remaining 41% manifests as functional interferences such as an IO error or a deadlock.

For example, **Hadoop-572** [26] (§I) is a vicious cycle caused by CPU contention between data recovery and heartbeat handling, resulting in a heartbeat delay. **HDFS-9107** [33] is a vicious cycle caused by contention between a GC during the DataNode's handling of FullBlockReport (FBR) requests and the sending of heartbeat messages.

The vicious cycle in **HDFS-12914** [32] originates from a performance bug delaying the handling of FBR requests and propagates through a deadlock on DataNode caused by HDFS block replications.

Implication: To expose and detect vicious cycles, it is critical to trigger performance and functional interferences between error handling logic and request handling logic. For example, injecting delays to health-checking (or degradation-checking) operations to trigger recovery and applying resource capping to induce performance interference could expose vicious cycles. Fault injection such as injecting IO errors is also useful in mimicking functional interferences. Monitoring techniques could focus on observing and correlating system degradation with performance degradation to detect many vicious cycles.

Finding 8: The interference causing a vicious cycle can result not only from a recovery task but also from **delaying** a recovery task.

Interestingly, not only a delay caused by a recovery task (e.g., CPU contention in Hadoop-572), a delay to the recovery task can also cause vicious cycles. Such a delay causes the error recovery information not being propagated to all the relevant nodes, so that other nodes handling requests can be interrupted by an error. **STORM-404** [39] is such a vicious cycle: In an Apache Storm cluster, when a worker W1 sends requests to another worker W2 but W2 unfortunately crashes. The error handler of the W2 crash on the coordinator relocates the tasks served on W2 to W3. However, such a topology change will only be sent to W1 when the coordinator receives a heartbeat from W1. For a substantial period of time, W1 incorrectly thinks W2 is still alive and keeps sending requests to W2. These requests are never terminated and crash W1 through a runtime exception. W2's status is misinterpreted by other workers who are sending requests to W2 and eventually further causing them to crash.

Implication: Both the recovery task and the **delay** of the recovery task could interfere with request handling, resulting in vicious cycles. Testing techniques could inject delays **before** the recovery task to expose potential vicious cycles.

D. UC Subtype 2: Unconstrained Retry

Finding 9: The last major (24%) reason for vicious cycles is unconstrained retries overloading the system, causing CPU (21%) and lock (3%) contentions, resulting in continuous request timeouts.

KAFKA-901 [37] is a vicious cycle caused by unconstrained retries. When a Kafka broker (node) is down, the clients will need to access the metadata of the cluster, which is slow and cause IO pressure on the nodes holding the metadata. Moreover, the threads handling the metadata request share the same thread pool with that handling the metadata response, and a spike in the request could delay the responses due to contention, which in turn causes more retries from the clients and would eventually make the cluster unavailable.

We find that commonly suggested prevention techniques such as exponential backoff are not universally applied to all retry procedures. However, as shown in our experiment (§VIII-B), applying exponential backoff can be a quick mitigation approach once the vicious cycle is discovered. Still, even for retry procedures with exponential backoff, it does not prevent vicious cycles completely because exponential backoff usually has a maximum backoff limit [89] and the same vicious cycle can still happen when the limit is reached.

Though unconstrained retry also introduces contention, it is different from vicious cycles caused by contention delaying the heartbeat. To prevent vicious cycles caused by unconstrained retry, the retry procedure should be confined to reduce the number or frequency of requests. In comparison, to prevent vicious cycles caused by contention delaying the heartbeat, the workload in each request handling logic should be reduced.

TABLE III: Triggering conditions of vicious cycles.

Trigger	%	Trigger	%
Heavy workloads	42%	Special cluster/node state	24%
Timing constraints	18%	Fault	21%
Input constraints	24%		

Implication: Unconstrained retries could easily overload the server if no backoff or other feedback-based prevention techniques are in place. The server can proactively drop user requests under a high load, and the client should refrain from future retries once the request is rejected. A simpler alternative is to use exponential backoff in non-latency-sensitive requests, but the stop condition of such techniques should be scrutinized to prevent vicious cycles.

VI. Triggering Conditions of Vicious Cycles

As discussed in §V and shown in the vicious cycle conceptual graph (Figure 2b), vicious cycles need special conditions to trigger. Table III shows an overview of the triggering conditions for vicious cycles. Note that we only consider the **necessary** triggering conditions for each vicious cycle to reduce the testing space for automated testing tools.

A. Special Constraints for Automated Testing Tools

Finding 10: About a quarter (24%) of the vicious cycles are triggered by inputs with special constraints. Many (21%) vicious cycles are triggered by faults in the cluster, including node failures, network disruptions, and out-of-memory errors.

As discussed in §V, many vicious cycles are caused by problematic retries that require special input constraints, including 1) inputs that trigger a timeout, and 2) inputs that trigger a latent logic bug or an error. For example, **FLINK-12342** [21] requires an allocation request for a large number of YARN containers, whose processing time exceeds the timeout limit.

Many vicious cycles start with a fault such as a node failure. For example, **Hadoop-572** [26] (§I) is triggered by a failure of a DataNode holding a large number of blocks. Although such a failure in turn triggers a heavy workload and violates the timing constraints of the heartbeat message, only the DataNode failure is **necessary** to trigger the bug.

Implication: Traditional testing techniques such as fuzz testing [50] and fault injection [67] can help explore the input constraints and faults to expose vicious cycles. Fuzzing should prioritize commands and options vulnerable to vicious cycles, such as commands that could induce a large workload and the options controlling timeout and retry behaviors.

Finding 11: 24% of the vicious cycles can only be triggered when the cluster is in a special state, such as during a rolling upgrade or when nodes are holding large amounts of data.

TABLE IV: Fixing strategies to vicious cycles [1].

Major redesign	%	Other Fix	%
Separation of heavy workload	6%	Adjusting timeout	9%
Workload reduction	18%	Using asynchronous operation	9%
System-specific	37%	Improving performance	6%
		Fixing logic errors	21%
Subtotal: Major redesign	61%	**Subtotal:** Other fix	42%
Workarounds	27%		

For example, **Hadoop-13738** [23] happens when the disks on many DataNodes in the cluster are full or nearly full. A logic error in the disk checker confuses the disk full with disk failures, causing a vicious cycle.

Implication: Though vicious-cycle-triggering states cannot be exhausted developers could leverage the special states revealed in our study during testing to expose vicious cycles.

B. Stress Testing is Useful to Trigger Vicious Cycles

Finding 12: A significant portion (42%) of vicious cycles require a heavy workload to trigger.

Implication: Resource contention is a major interference caused by error handlers (§V). Proactively limiting available resources and stress testing are useful to trigger many vicious cycles. Also, testing the systems' behavior after and beyond the overloading point is important, as the failure recovery afterwards can still trigger vicious cycles.

C. Vicious Cycles Can Be Non-deterministic

Finding 13: A large portion (36%) of vicious cycles are non-deterministic, and a non-negligible amount (18%) requires timing constraints with very small time windows.

Exposing concurrency bugs in distributed software systems with concurrency testing [76], [78] and model checking [73], [83], [93] is extremely challenging. A combination of heavy workloads and small timing windows creates a large testing space, which challenges existing automated testing techniques.

For example, as discussed in §8, **STORM-404** [39] is a vicious cycle that requires a fair amount of requests and a small time window. Specifically, since the delayed knowledge of the node failure directly leads to the vicious cycle, the only time window available to trigger the bug is one heartbeat interval, during which worker W2 has crashed but worker W1 has not learned about the node failure yet.

VII. Fix Strategies for Vicious Cycles

We find that fixes to vicious cycles are highly complex and ad hoc. Table IV shows an overview of the fixing strategies for vicious cycles. There are two major fixing categories: 1) major redesign and 2) other fixes. The workaround row shows the percentage of fixes that are considered as a workaround instead of a complete fix by the developers themselves.

[1] Numbers do not add to 100%, because some vicious cycles are fixed by a combination of multiple fixing strategies.

A. Patterns of Major System Redesigns

Finding 14: A significant portion (61%) of vicious cycles are fixed by major redesigns, including 1) separating heavy workload (6%), 2) reducing the workload (18%), and 3) other system-specific redesigns (37%).

To fix **Hadoop-572** [26] (§I), the developers reduce the workload (generating block replication commands) in each heartbeat message by demanding a block report from any resurrected DataNodes. When a DataNode is temporarily marked dead at the NameNode side and later reconnects, the blocks available on that DataNode could be immediately available after the block report, thus greatly reducing the need for generating block replication commands.

However, the fixes introduced in Hadoop-572 are not complete, and the same bug is captured again in a later JIRA ticket: **Hadoop-923** [27]. It is fixed by a major redesign of the heartbeat handling logic, separating the unbounded workload - generating block replication requests - out of the heartbeat reply and creating a new command for this function. Combined with a better scheduling policy (e.g., FairCallQueue [19]), such a vicious cycle caused by contention is fixed.

Implication: Any heavy workload in the distributed software systems should be measured for the latency and the overall turnaround time to avoid contention with time-sensitive operations such as the heartbeat. Such workloads can also be processed with **a separate command or thread pool** asynchronously. A better scheduling policy, such as prioritizing the heartbeat processing is also a good practice.

B. Other Fixing Patterns of Vicious Cycles

Finding 15: 42% of vicious cycles are fixed using approaches other than a redesign, including adjusting the timeout limits (9%), utilizing asynchronous requests (9%), applying performance optimization (6%), and fixing logic errors (21%).

Replacing a synchronous operation with its corresponding asynchronous version is a general pattern for fixing vicious cycles. For example, **ZooKeeper-1049** [47] is fixed by making the TCP close() calls asynchronous. So that they do not block the heartbeat handlers in the same thread pool. Such a fix is considered a general pattern because it does not change the logic or the architecture of the system, compared to the separation of heavy workloads, which changes request handling logic. Similarly, adjusting timeout is also a common practice, though developers often use it as a short-term workaround (§VII-C). Applying performance optimization and fixing logic errors are typically system-specific.

C. Vicious Cycles Often Have Incomplete Fixes

Finding 16: More than a quarter (27%) of fixes are considered incomplete by developers.

As the Hadoop-572 discussed above, vicious cycles are hard to fix. Though the root causes for each bug can be correctly identified by the developers, the fixes proposed in each bug ticket can still be incomplete, i.e., it does not prevent future occurrences of a similar vicious cycle.

To get an objective result, we conservatively use the developers' assessment to determine a fix's completeness. If the fix patch is later admitted by the developers as a "short-term solution", "workarounds", or "mitigation", we consider the patch incomplete. Using this criterion, we find that 27% of the fixes are incomplete, and the bug Hadoop-572 mentioned above is one of them, which is observed again in Hadoop-923.

Implication: To determine the completeness of the fixes, vicious cycles can be revisited by triggering the original buggy execution. For example, if only the timeout limit is adjusted or a performance improvement is applied, developers can conduct stress testing again on the patched system.

A previous study [60] found that 12% of the failure recovery bugs are not fixed completely. Our study indicates that vicious cycles are harder to be fixed than general failure recovery bugs, especially noting the fact that we only consider developers' direct confirmation of utilizing workarounds.

VIII. Preventing Vicious Cycles: A Feasibility Study

We investigate the feasibility of preventing vicious cycles by 1) building a prototype tool that prevents deadly retries and 2) implementing a commonly suggested practice – exponential backoff – to prevent unconstrained retries. They account for 45% of the vicious cycles we studied. Their solutions also facilitate concrete discussions for more general solutions that apply to other types of vicious cycles. The limitations of our feasibility study are further discussed in §IX.

A. Preventing Deadly Retry

Design. The key to preventing deadly retry is to identify the causal relationship between the error-inducing retried request and the error (§V-B), This helps the system determine whom to blame and block the error-inducing request. Our tool identifies this causal relationship with a central monitor which correlates repeated requests with repeated errors on multiple nodes.

For each node in the cluster, we deploy an agent to record recent outgoing RPC requests. The agent also forwards any error (e.g., ERROR and FATAL log messages, and crash) to the central monitor. The monitor collects such errors from each node and analyzes them to detect repeated errors across nodes. Once a repeated error is detected, the monitor broadcasts a message containing the set of nodes (S) on which the error is repeated to the entire cluster. Each node replies with its recent outgoing RPC requests sent to the nodes in S. Upon receiving these RPC requests, the monitor calculates their signatures to identify the repeated requests sent to S and instruct the agents to block further retries of these requests.

Implementation. We use AspectJ [71] to instrument the RPC library, and gRPC [22] to implement the agent-monitor communication. Each agent is implemented as an extra daemon thread, and all the RPC messages are recorded in an in-memory FIFO queue with a configurable fixed size.

TABLE V: Result on reproduced deadly retry vicious cycles. A tick in column *P.?* indicates that the vicious cycle is prevented. Column "Reason" explains why the vicious cycle is not prevented.

Vicious Cycle	P.?	Reason
HBase-14598 [28]	✓	
HBase-23076 [29]	✓	
Flink-10928 [20]		The error frequency of the vicious cycle is low.

We use the string representation of RPC message content to identify retried RPC requests. To accommodate the potential variance of message content when the erroneous request is retried on different nodes, we identify a retried RPC message by calculating its edit distance [86] to the request signatures calculated by the monitor. If the string representation of the current outgoing RPC request is more than 90% similar to the signature, the agent will block this request from sending.

A caveat of our approach is batched RPC requests. We leverage the `repeated` field modifier in Protocol Buffers [38] to identify a batched request. If the string representation of the `repeated` fields makes up more than 85% of the entire RPC message content, we consider the request as a batched RPC request. The batched RPC requests are then split into multiple sub-requests, and the identification of retried erroneous requests is carried out at the sub-request level.

Though constructing string representations and computing edit distances are expensive, they are only performed when a repeated error is detected.

Evaluation. We evaluate our tool on all the vicious cycles caused by deadly retry that we can reproduce. Table V shows that our tool successfully prevents HBase-14598 [28] and HBase-23076 [29] without interfering with the normal functionality of the system.

Flink-10928 [20] cannot be prevented because it involves an OOM error that happens repeatedly but infrequently. If the FIFO queue size is too small, we cannot retrieve the retried request because it may have already been evicted. If the size is too large, the false positive rate will increase. The root cause of the OOM error is a memory leak. In the future, one may integrate a runtime memory leak detection tool [53], [54], [80] with our solution to prevent such vicious cycles.

The remaining four cases are not reproduced due to 1) the non-determinism nature of the bug, 2) lack of information in the bug ticket, and 3) incompatible JDK version requirements.

Impact on normal executions and limitation. We evaluate the overhead of our solution with six standard benchmarks from the "Yahoo! Cloud Serving Benchmark" [55] (YCSB), which is a popular benchmark for evaluating the performance of distributed software systems [57], [58], [66], [91]. Each of the benchmarks is run 5 times. The introduced overhead is 0.31% on average with a maximum of 0.98%.

It is possible that our solution incorrectly blocks non-error-inducing requests if it incorrectly associates a normal request with an error (false positive). However, we do not observe such

Fig. 3: An example of retry loop from Hadoop 2.4.0 [25].

TABLE VI: Result on reproduced unconstrained retry vicious cycles. A tick in column *P.?* indicates that the vicious cycle is prevented. Column "Reason" explains why the vicious cycle is not prevented.

Vicious Cycle	P.?	Reason
Hadoop-16284 [24]	✓	
HBase-27149 [30]	✓	
Kafka-6028 [36]	✓	
Flink-12342 [21]		Request piggybacked on heartbeat

cases in our evaluation. To balance between false positive and false negative rates, three parameters in the implementation are selected based on empirical experiments (thus, limited to HBase and Flink): 1) the size of the FIFO queue, 2) the threshold identifying the request to block, and 3) the threshold identifying batched requests.

Discussion. Our tool shows the feasibility of preventing vicious cycles by making a more informed decision in error handling. It achieves this by collecting one type of error context – recent RPC requests – with low overhead. Preventing other vicious cycles (e.g., those that happen when the error handler cannot distinguish error induced by an external trigger and the interference of error recovery) requires different error contexts to be collected (e.g., information about the interference). In addition, the selected parameters need to be tuned for more systems and, if needed, automatically adjusted to achieve a good balance between false positive and false negative rates.

B. Preventing Unconstrained Retry

Figure 3 shows an excerpt of a retry loop of RPC invocation in Hadoop [25]. The retry logic manifests as an infinite loop (L2), and the RPC invocation is enclosed by a try-catch block (L3). If an exception is thrown during the RPC invocation, a RetryPolicy (L6 and L7) decides 1) whether the request should be retried further (if not, rethrow the exception on L6), 2) how long should the thread wait until the next retry (L8).

After reproducing each vicious cycle, we manually inspect the source code and identify the loops that can cause unconstrained retries. Then we inject exponential backoffs at the loops to test if the vicious cycle can be prevented. Our evaluation shows the feasibility of using exponential backoff to prevent vicious cycles. One promising future direction is to automate the process of identifying retry-inducing loops statically using patterns such as the one shown in Figure 3.

Evaluation. As in Table VI, injecting exponential backoffs can prevent 75% of the vicious cycles reproduced. This approach cannot address Flink-12342 [21] as the retried request is piggybacked on the heartbeat, which cannot be delayed.

Impact on normal executions and limitation. Injecting exponential backoff may affect the correctness of the system because picking the optimal backoff parameters is challenging due to dependencies in timeout configurations. For example, in Hadoop-16284 [24], the retried request first lands on a server-side cache, when the backoff cap is set too large, the retried request always experiences cache miss, because the retrieved cache entry is evicted (timed out) before the next retry. In our experiments, we carefully inspect the source code and configuration file to make sure that the correctness of the system is not affected.

Discussion. For vicious cycles induced by performance interference, exponential backoff is an effective solution as long as the recovery task can be delayed. However, picking backoff parameters safely without affecting the system's correctness can be challenging. In addition, injecting exponential backoff on the client side could be undesired due to latency constraints. To automatically inject exponential backoff and prevent vicious cycles induced by performance interference, automatically identifying backoff locations, automatically setting safe backoff parameters, and dynamically adjusting backoff parameters to reduce latency are important future directions to explore.

IX. Threats to Validity and Limitations

Representativeness of bug reports from the issue tracker. There could be vicious cycles not reported to issue trackers. For instance, vicious cycles formed due to resource contention could be mitigated by changing configurations or adding resources. A mailing list can provide such solutions.

Limitations of the filtering criteria. We could have missed vicious cycles whose issue reports do not contain our selected keywords for two reasons. 1) A vicious cycle could be treated as a symptom while the underlying bug forming the cycle is considered the root cause. Many reports on issue trackers mainly focus on the root causes, searching with keywords describing the symptom could result in false negatives. 2) Our multi-round boosting strategy is a best-effort strategy and could have missed some keywords.

Representativeness of selected distributed software systems. Our study does not include vicious cycles in closed-source systems and they could have different characteristics. Our study focuses on open-source software systems because we favor a source-code-level understanding of vicious cycles.

Possible observer errors. There is a risk of observer errors. To minimize the effect, each failure was investigated by at least two inspectors with the same detailed criteria. Any disagreement is discussed in the end to reach a consensus.

Limited number of vicious cycles investigated. The generalizability of our findings may be affected by the number of vicious cycles included in our study. Our filtering process of the bug reports may miss some vicious cycles as many reports focus on the root cause and proposed fix, rather than detailing symptoms and logs. However, our findings cover a wide range of open-source distributed software systems on various workloads (§III). This indicates that the findings are workload-agnostic and highlights the prevalence of vicious cycles in open-source systems.

Limitations of the feasibility study. Our feasibility study is based on a limited number of reproduced vicious cycles related to the retry behavior of the system. For the unreproducible retry-related vicious cycles due to 1) the non-deterministic nature of the bug, 2) lack of information in the bug report, and 3) incompatible JDK or dependency version, we cannot evaluate the effectiveness of our tool on them. The feasibility study also does not cover the non-retry-related vicious cycles due to their diversity in symptoms and programming paradigms. For example, the bug in Cassandra-13441 (§V-A) is system-specific, and the bug in HDFS-12914 (§V-C) is performance related. In addition, as explained in Section VIII, the effectiveness of our tools is affected by the parameters empirically selected.

X. Related Work

Distributed Software System Failure Study. Many studies [48], [49], [59], [62]–[64], [69], [74], [79], [81], [85], [87], [88], [94], [97] have analyzed distributed software system failures. Gunawi et al. [62] conducted a comprehensive study of 3,655 high-priority issues in widely-used open-source distributed software systems. Unfortunately, only one of the 3,655 issues [62] contains a vicious cycle, highlighting the difficulty of gathering such issues from open-source systems.

There has been a growing interest in studying and addressing cascading failures, such as incorrect failure recovery [65], cascading performance bugs [77], metastable failures [51], [68], and cascading virtual machine failure [92]. However, none of them defined and discussed vicious cycles in detail.

Closest to our study are recent works on the metastable failure state [51], [68] when a system is under permanent overload due to a work amplification. While insightful, only resource contention and unconstrained retry are studied as the root cause. Our study reveals vicious cycles are much more diverse. Also, all 21 metastable failures were from proprietary cloud systems such as Amazon AWS, which hinders their reproduction and analysis for future research.

Unlike Huang et al.'s [68] observation of high-level system behavior, our definition of vicious cycles focuses on the execution trace of the system. Our approach provides a detailed understanding of the system behavior, especially the behavior of the error handler, which is useful for developing automated testing and runtime monitoring tools.

Error handlers, such as exception handlers and crash recovery procedures, have garnered substantial research interest [52], [60], [95] in recent years. However, none of them study vicious cycles. Our study complements these studies by analyzing vicious cycles in detail and reveals the reason why error handlers can lead to vicious cycles: the inability to distinguish errors caused by an external trigger and the interference due to lack of information (§V). Our study also reveals that vicious cycles do not always involve explicit errors (e.g., exceptions and crashes) and incorrect handlers, but sometimes happen due to silent errors and missing handlers (§V-A). In addition, our bug dataset [45] does not overlap with the crash recovery bug [16] and exception-related bug datasets [18].

Guo et al. [65] provided a case study of 4 cascading failures that happen in the failure recovery stage in production commercial clusters. Our study corroborates their result. On the other hand, our study provides a deeper analysis of vicious cycles in open-source systems and proposes actionable suggestions to combat vicious cycles.

Testing Tools. Recent works [73], [77], [82], [83], [90], [93] scaled automated system-level testing to distributed software systems by focusing the test effort on specific bug patterns. However, vicious cycles have yet to be covered.

PCatch [77] detects performance cascading bugs by utilizing a causal relationships model [78]. Performance cascading bugs happen when a large user job delays another user job, but does not necessarily involve system degradation. In comparison, vicious cycles focus on the aggravating cycle between the system's execution, including both user requests and internal requests, and system degradation.

CrashTuner [82] utilizes log-guided fault injection [82] to automatically test for concurrency bugs that happen during failure recovery. Our study shows most vicious cycles are not caused by concurrency bugs.

ScaleCheck [90] triggers scalability bugs by injecting sleep in CPU-intensive computations that involve scale-dependent data structures. Though some vicious cycles (e.g., Hadoop-572) are easier to be triggered in a large-scale cluster, most vicious cycles do not have such a requirement.

XI. Conclusion

This paper provides the first in-depth analysis of vicious cycles in open-source distributed software systems. To overcome the challenge of collecting issues containing vicious cycles, we adopt a multi-round boosting strategy to expand our filtering criteria until it converges. We further analyze the symptoms, root causes, triggering conditions, and the fixing strategies of vicious cycles. Based on the findings, we present two approaches that can prevent vicious cycles caused by retries. Our study reveals 16 findings with concrete implications. In particular, we discuss implications and guidelines for runtime detection and prevention techniques, testing techniques, and good practices in fix and development strategies to avoid vicious cycles.

Acknowledgments

The authors thank anonymous reviewers for their constructive comments. They thank Tian Xie for examining some bug reports. This research is partially supported by NSF 1901242 and 2006688. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

References

[1] "Apache Accumulo," https://accumulo.apache.org/.
[2] "Apache ActiveMQ," https://activemq.apache.org/.
[3] "Apache Cassandra," https://cassandra.apache.org/.
[4] "Apache Flink," https://flink.apache.org/.
[5] "Apache Hadoop," https://hadoop.apache.org/.
[6] "Apache hbase," https://hbase.apache.org/.
[7] "Apache Ignite," https://ignite.apache.org/.
[8] "Apache Kafka," https://kafka.apache.org/.
[9] "Apache Ratis," https://ratis.apache.org/.

[10] "Apache Solr," https://solr.apache.org/.
[11] "Apache Storm," https://storm.apache.org/.
[12] "Apache ZooKeeper," https://zookeeper.apache.org/.
[13] "AWS Post-Event Summaries," https://aws.amazon.com/premiumsupport/technology/pes/.
[14] "Cassandra-13441," https://issues.apache.org/jira/browse/CASSANDRA-13441.
[15] "A conjecture on why reliable systems fail – surfing complexity," https://surfingcomplexity.blog/2017/06/24/a-conjecture-on-why-reliable-systems-fail/.
[16] "CREB Dataset," http://www.tcse.cn/~wsdou/project/CREB/.
[17] "Data replication," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Data_Replication.
[18] "eBugs in cloud systems," https://hanseychen.github.io/eBugs/.
[19] "Fair call queue guide," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FairCallQueue.html.
[20] "Flink-10928," https://issues.apache.org/jira/browse/FLINK-10928.
[21] "Flink-12342," https://issues.apache.org/jira/browse/FLINK-12342.
[22] "gRPC: A high performance, open source universal RPC framework," https://grpc.io/.
[23] "Hadoop-13738," https://issues.apache.org/jira/browse/HADOOP-13738.
[24] "Hadoop-16284," https://issues.apache.org/jira/browse/HADOOP-16284.
[25] "Hadoop 2.4.0," https://github.com/apache/hadoop/blob/a0389ac6aa83d8ec8dc8dbe22fd423e8a93c2c62/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/io/retry/RetryInvocationHandler.java#L78.
[26] "Hadoop-572," https://issues.apache.org/jira/browse/HADOOP-572.
[27] "Hadoop-923," https://issues.apache.org/jira/browse/HADOOP-923.
[28] "HBase-14598," https://issues.apache.org/jira/browse/HBASE-14598.
[29] "HBase-23076," https://issues.apache.org/jira/browse/HBASE-23076.
[30] "HBase-27149," https://issues.apache.org/jira/browse/HBASE-27149.
[31] "HBase Cluster Replication," https://hbase.apache.org/book.html#_cluster_replication.
[32] "HDFS-12914," https://issues.apache.org/jira/browse/HDFS-12914.
[33] "HDFS-9107," https://issues.apache.org/jira/browse/HDFS-9107.
[34] "HDFS architecture guide," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[35] "Kafka-10888," https://issues.apache.org/jira/browse/KAFKA-10888.
[36] "Kafka-6028," https://issues.apache.org/jira/browse/KAFKA-6028.
[37] "Kafka-901," https://issues.apache.org/jira/browse/KAFKA-901.
[38] "Protocol Buffers Documentation," https://protobuf.dev/.
[39] "Storm-404," https://issues.apache.org/jira/browse/STORM-404.
[40] "Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region," https://aws.amazon.com/message/5467D2/.
[41] "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region," https://aws.amazon.com/message/65648/.
[42] "Summary of the Amazon SimpleDB Service Disruption," https://aws.amazon.com/message/65649/.
[43] "Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region," https://aws.amazon.com/message/12721/.
[44] "Summary of the October 22, 2012 AWS Service Event in the US-East Region," https://aws.amazon.com/message/680342/.
[45] "Vicious Cycle Study Dataset," https://github.com/lin-tan/vcstudy.
[46] "YARN ResourceManager HA," https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html.
[47] "Zookeeper-1049," https://issues.apache.org/jira/browse/ZOOKEEPER-1049.
[48] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany, "Toward a Generic Fault Tolerance Technique for Partial Network Partitioning," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.
[49] 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 Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.
[50] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *2013 35th International Conference on Software Engineering (ICSE'13)*, 2013.
[51] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, "Metastable failures in distributed systems," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*, 2021.
[52] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding Exception-Related Bugs in Large-Scale Cloud Systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*, 2019.
[53] Z. Chen, C. Wang, J. Yan, Y. Sui, and J. Xue, "Runtime detection of memory errors with smart status," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 296–308.
[54] Z. Chen, J. Yan, S. Kan, J. Qian, and J. Xue, "Detecting memory errors at runtime with source-level instrumentation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 341–351.
[55] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
[56] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC'87)*, 1987.
[57] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 14)*, 2014, pp. 401–414.
[58] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 12)*, 2012, p. 37–48.
[59] 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 Systems Design and Implementation (OSDI'10)*, 2010.
[60] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*, 2018.
[61] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.
[62] 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)*, 2014.
[63] 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)*, 2016.
[64] 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)*, 2018.
[65] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou, "Failure recovery: When the cure is worse than the disease," in *14th Workshop on Hot Topics in Operating Systems (HotOS'13)*, 2013.
[66] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa, "A statistics-based performance testing methodology for cloud applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, 2019, pp. 188–199.
[67] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, April 1997.
[68] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko, "Metastable failures in the

wild," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022.

[69] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, 2017.

[70] A.-M. Kermarrec and M. Van Steen, "Gossiping in distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 5, pp. 2–7, October 2007.

[71] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP 2001—Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15*. Springer, 2001, pp. 327–354.

[72] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, July 1978.

[73] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[74] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, 2016.

[75] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990 (ICSM'90)*, 1990.

[76] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[77] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "Pcatch: Automatically detecting performance cascading bugs in cloud systems," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*, 2018.

[78] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[79] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What Bugs Cause Production Cloud Incidents?" in *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*, 2019.

[80] C. Lou, C. Chen, P. Huang, Y. Dang, S. Qin, X. Yang, X. Li, Q. Lin, and M. Chintalapati, "RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022.

[81] C. Lou, P. Huang, and S. Smith, "Understanding, Detecting and Localizing Partial Failures in Large System Software," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, 2020.

[82] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[83] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa *et al.*, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19)*, 2019.

[84] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[85] B. Maurer, "Fail at Scale: Reliability in the Face of Rapid Change," *Communications of the ACM*, vol. 58, no. 11, pp. 44–49, November 2015.

[86] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[87] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

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

[89] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "RFC3261: SIP: Session Initiation Protocol," USA, 2002.

[90] C. A. Stuardo, T. Leesatapornwongsa, R. O. Suminto, H. Ke, J. F. Lukman, W.-C. Chuang, S. Lu, and H. S. Gunawi, "ScaleCheck: A Single-Machine approach for discovering scalability bugs in large distributed systems," in *17th USENIX Conference on File and Storage Technologies (FAST'19)*, 2019.

[91] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD' 17)*, 2017, p. 1009–1024.

[92] H. Wang, H. Shen, and Z. Li, "Approaches for resilience against cascading failures in cloud datacenters," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*, 2018.

[93] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: Transparent model checking of unmodified distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[94] 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)*, 2014.

[95] D. Yuan, Y. Luo, X. Zhuang, G. R. 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[96] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[97] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and Detecting Software Upgrade Failures in Distributed Systems," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, 2021.