



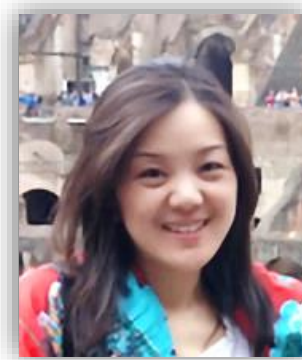
# Vicious Cycles in Distributed Software Systems



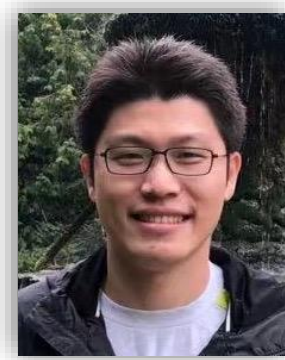
**Shangshu Qian**



Wen Fan



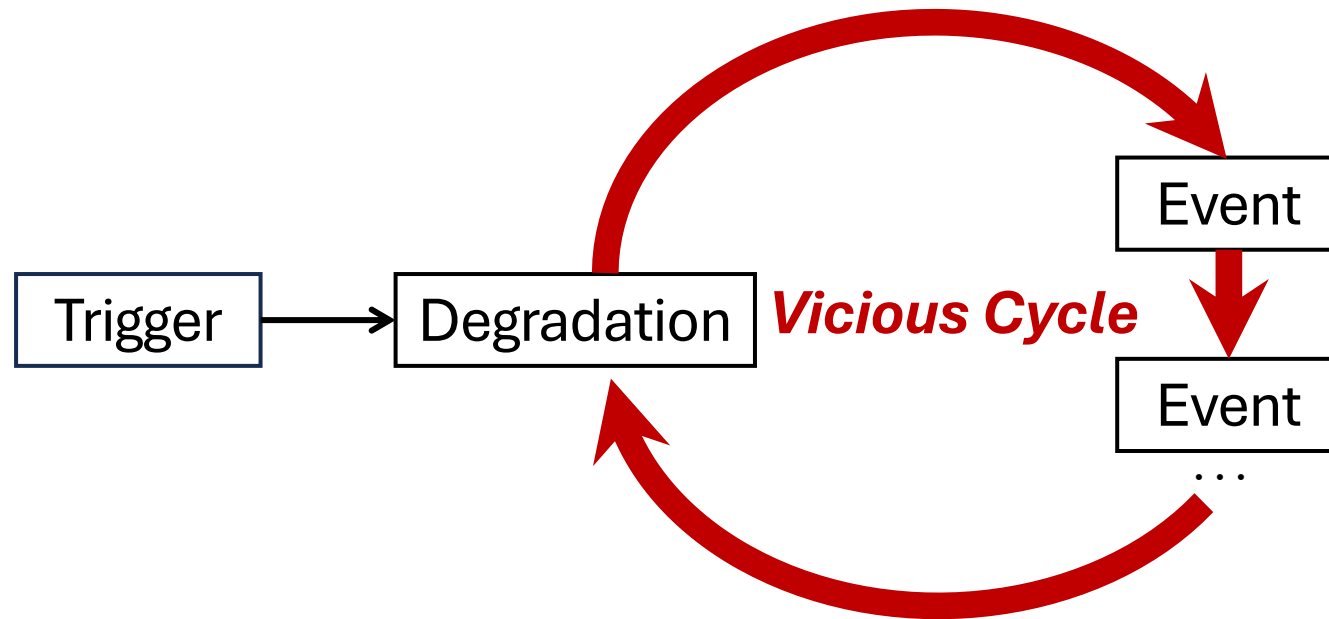
Lin Tan



Yongle Zhang

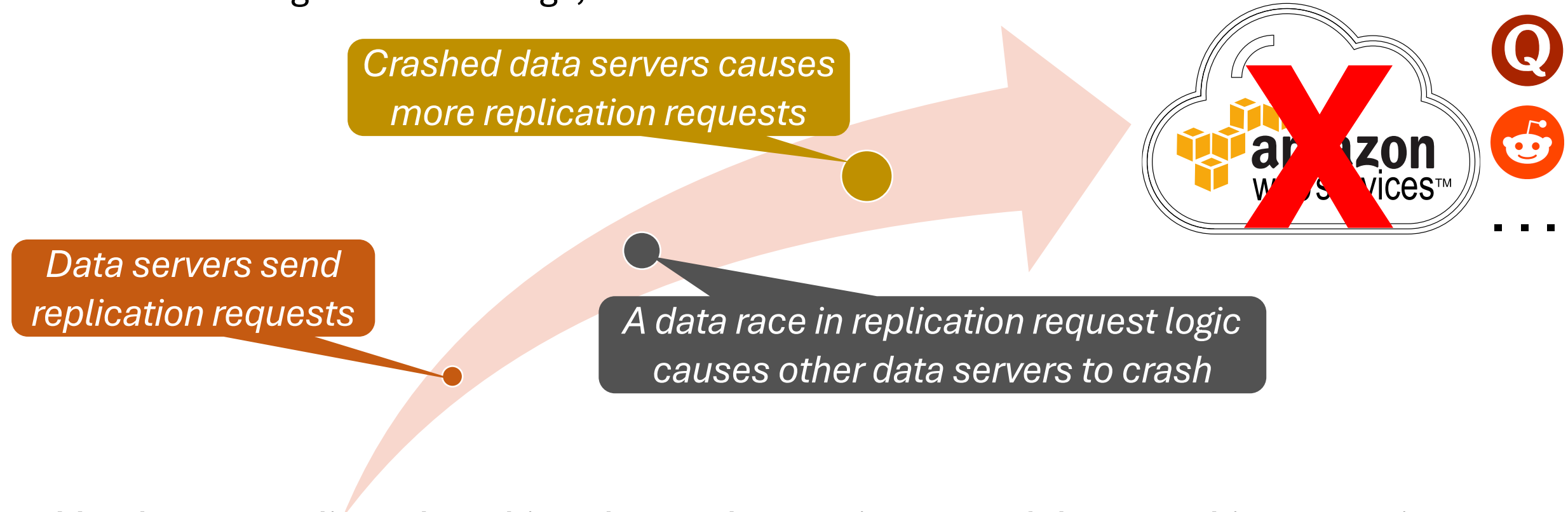
# Vicious Cycle: Self-Reinforcing Failures in Distributed Systems

- Vicious Cycle: A **self-reinforcing** cycle between events and system degradation during a distributed system's execution.
  - Degradation: e.g., node crash, missing data blocks
  - Event: Any other normal execution in the system



# Vicious Cycles are Catastrophic!

- Large-scale impact: the cycle propagates across nodes.
  - AWS Storage Service outage, 2011.



- Hard to remediate: breaking the cycle requires careful manual intervention.
  - AWS outage took 3 days to recover: fine-grained throttling, adding physical servers.

# Vicious Cycles are a Major Reason for Cloud Outage

- One third of the **most catastrophic AWS** outages involve vicious cycles.
  - Most recent one in 2021.

Summary of the AWS Service Event in the Northern Virginia (US-

Summary of the Amazon DynamoDB Service Disruption and Related Impacts

Summary of the Amazon SimpleDB Service Disruption

Summary of the October 22, 2012 AWS Service Event in the US-East Region

Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region

April 29, 2011

Now that we have fully restored functionality to all affected services, we would like to share more details with our customers about the events that occurred with the Amazon Elastic Compute Cloud ("EC2") last week, our efforts to restore the services, and what we are doing to prevent this sort of issue from happening again. We are very aware that many of our customers were significantly impacted by this event, and as with any significant service issue, our intention is to share the details of what happened and how we will improve the service for our customers.

The issues affecting EC2 customers last week primarily involved a subset of the Amazon Elastic Block Store ("EBS") volumes in a single Availability Zone within the US East Region that became unable to service read and write operations. In this document, we will refer to these as "stuck" volumes. This caused instances trying to use these affected volumes to also get "stuck" when they attempted to read or write to them. In order to restore these volumes and stabilize the EBS cluster in that Availability Zone, we disabled all control APIs (e.g. Create Volume, Attach Volume, Detach Volume, and Create Snapshot) for EBS in the affected Availability Zone for much of the duration of the event. For two periods during the first day of the issue, the

"Once a system reaches a certain level of reliability, most major incidents involve a self-reinforcing cycle." [1]

# Vicious Cycles are Under-Investigated

- Existing studies are limited
  - Guo et al. <sup>[1]</sup> investigated four vicious cycles
  - Huang et al. <sup>[2]</sup> focuses on contention-induced vicious cycles
  - Both performed on proprietary cloud systems

[1] Guo, Zhenyu, et al. "Failure recovery: When the cure is worse than the disease." 14th Workshop on Hot Topics in Operating Systems (HotOS XIV). 2013.

[2] Huang, Lexiang, et al. "Metastable failures in the wild." 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 2022.

# First Empirical Study on Vicious Cycles in Open-Source D.S.

- **33 Vicious Cycles, on 13 open-source distributed software systems**
  - Studied systems: HDFS, Hadoop, HBase, Cassandra, Kafka, etc.
  - Case collection: Keyword-based searching with manual filtering
  - Source-code-level study with 8 bugs reproduced for better understanding
    - Each one takes one week on average

# Contribution

- 16 findings revealing unique characteristics of vicious cycles
  - Symptom
    - Majority (55%) of vicious cycles' degradation grows exponentially, while almost half grows linearly.
  - Root cause
    - **Key insight:** Vicious cycles are formed due to **missing** or **insufficiently informed** error handlers.
  - Triggering conditions
  - Fixing strategies
- Feasibility study: Automatically detecting & preventing vicious cycles
  - Feeding error handlers with required causal information
- The bug dataset, detailed analysis, and code are publicly available<sup>[1]</sup>.

[1] <https://github.com/lin-tan/vcstudy>

# Root Cause: How are Vicious Cycles Formed?

**Insufficiently Informed Error Handlers**

Vicious Cycle Type	Subtype	Interference
<b>Unexpected Cycle</b> (60%)	Incorrect Degradation Recovery (36%)	Performance (21%) Functional (15%)
	Unconstrained Retry (24%)	Performance (24%)
<b>Unexpected Error</b> (40%)	Undetected Error (18%)	N/A
	Unhandled Error (22%)	N/A

**Missing Error Handlers**



# Root Cause: How are Vicious Cycles Formed?

**Insufficiently Informed Error Handlers**

Vicious Cycle Type	Subtype	Interference
Unexpected Cycle (60%)	<b>Incorrect Degradation Recovery (36%)</b>	Performance (21%) Functional (15%)
	<b>Unconstrained Retry (24%)</b>	Performance (24%)
	Undetected Error (18%)	N/A
Unexpected Error (40%)	Unhandled Error (22%)	N/A

**Missing Error Handlers**

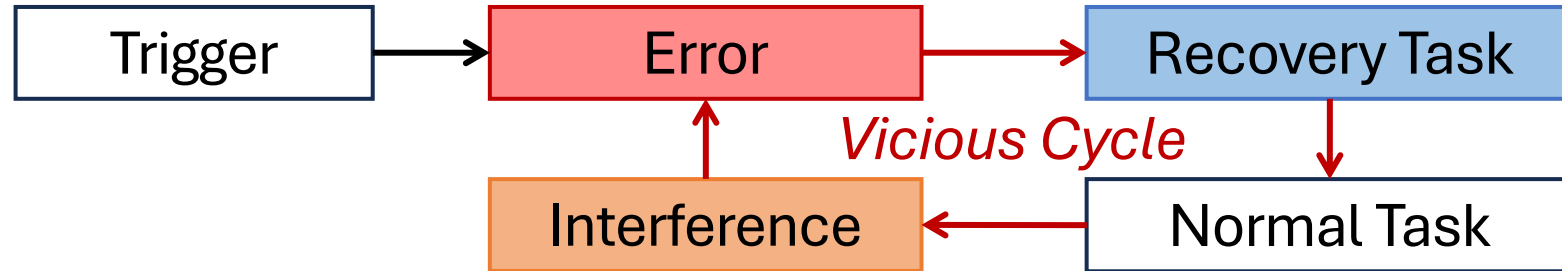
# Root Cause: How are Vicious Cycles Formed?

**Insufficiently Informed Error Handlers**

Vicious Cycle Type	Subtype	Interference
Unexpected Cycle (60%)	Incorrect Degradation Recovery (36%)	Performance (21%) Functional (15%)
	Unconstrained Retry (24%)	Performance (24%)
	<b>Undetected Error (18%)</b>	N/A
Unexpected Error (40%)	<b>Unhandled Error (22%)</b>	N/A

**Missing Error Handlers**

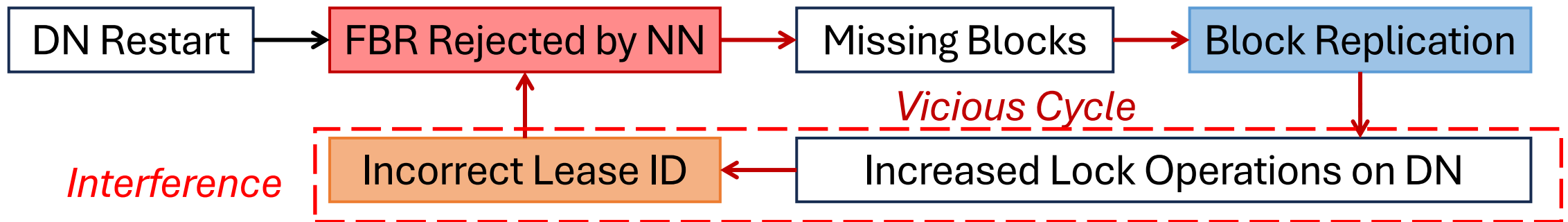
# Unexpected Cycle: Insufficiently Informed Error Handler



- An external trigger causes an **error**.
- The **recovery task** performed by the error handler unexpectedly **interferes** with the request handling and causes other requests to fail.
  - Functional interference: IO error, deadlock, data race, etc.
  - Performance interference: CPU, memory, and network contention

# Incorrect Degradation Recovery w/ Functional Interference

- HDFS-12914: HDFS loses a large number of DataNodes after restart.



- *DN: DataNode*
- *FBR: Full Block Report (part of DN registration after restart)*
- *NN: NameNode*

# Unexpected Cycle Example 2: Performance Interference

---

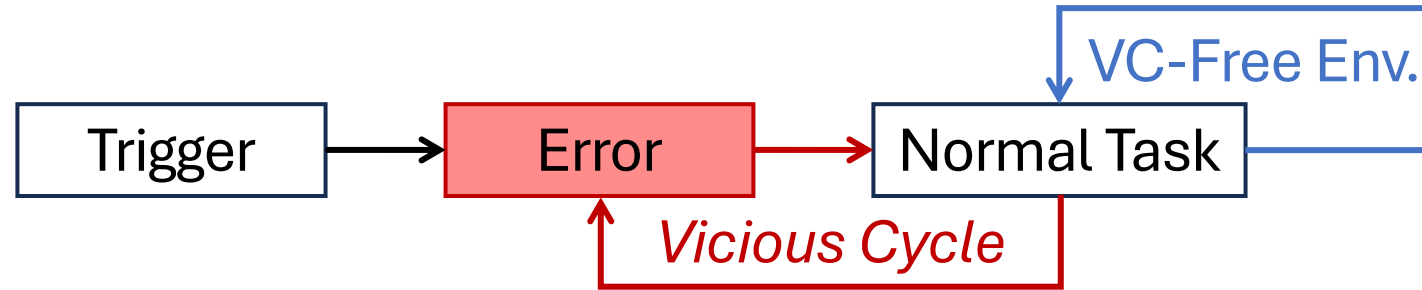
Unexpected Cycle (60%)	Incorrect Degradation Recovery (36%)	Performance (21%)
		Functional (15%)
	<b>Unconstrained Retry (24%)</b>	<b>Performance (24%)</b>

---

- Feasibility study 1: Applying exponential backoff

More details in the paper!

# Unexpected Error's Propagation Along a Global Cycle



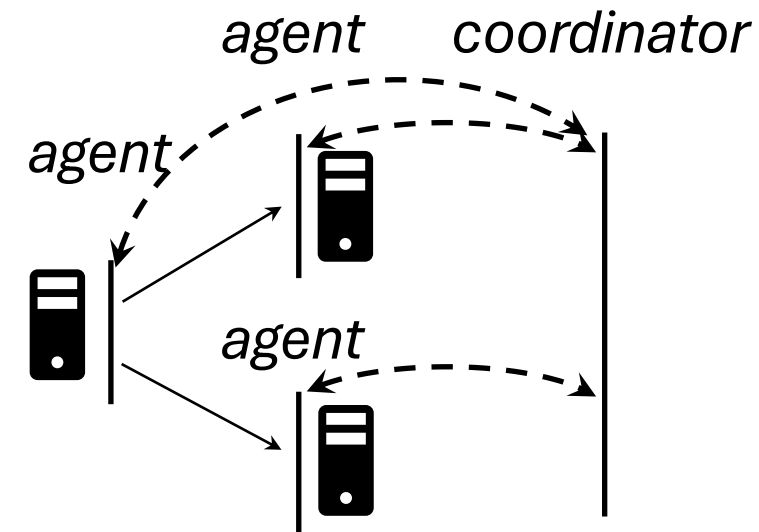
- An error that hinders the task completion is propagated along a global cycle.
  - Unhandled Error: The error in the cycle is observed, but not properly handled.
  - Undetected Error: The error in the cycle is silent, thus, no error handler is implemented.

# UE Subtype 1: Deadly Retry of Unhandled Errors

- Retry error-inducing requests on multiple nodes, causing them to fail.
- Root cause: missing causal information
  - Unable to infer the causality between the **error** and the **error-inducing retried request**

# UE Subtype 1: Deadly Retry of Unhandled Errors

- Feasibility study 2: Preventing deadly retries
  - Infer causality between error and request
  - Local agent: Records RPC requests and fatal errors.
  - Central coordinator
    - Monitors repeated fatal errors from different nodes
    - Compares recent RPCs from failed nodes to identify the error-inducing retried request, and blocks the request
  - Result: Successfully prevents two vicious cycles





# UE Subtype 2: Undetected Error Causing Vicious Cycles

- Vicious cycle: Undetected error spread by a normal loop execution.
  - Reason: Error detector fails to distinguish an error from a normal execution.
  - Majority (83%) are caused by logic errors: hard to detect automatically.

# Potential Solutions

- **Detect & break** the cycle: Informed recovery decision
  - Infer the causal relationship between requests and errors
- **Test** for the cycle: Trigger the interference
  - Targeted fault injection to trigger the interference between error handler & request handler

We need better error handlers with **rich feedback information!**

*More findings and lessons in the paper!*