Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States

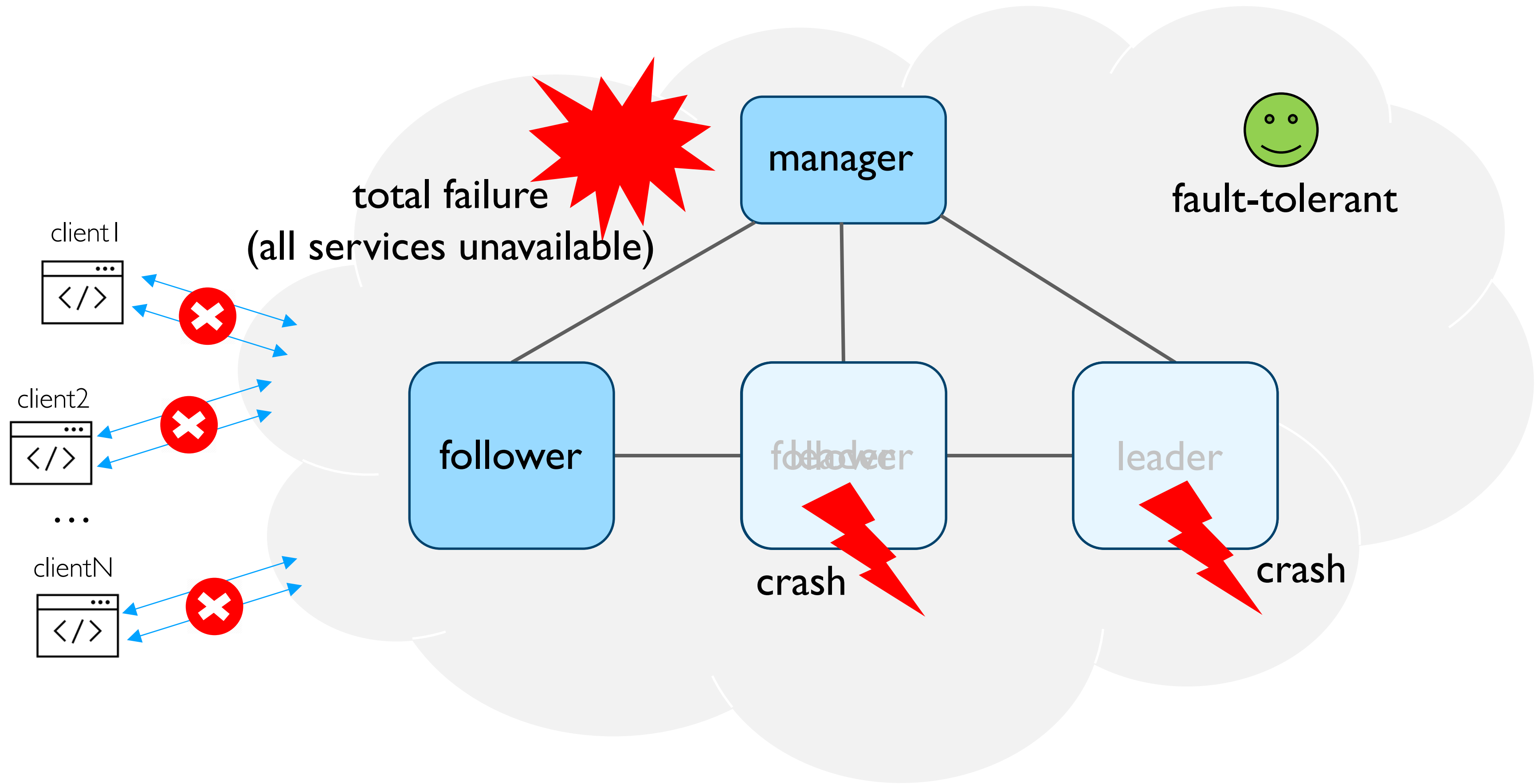
Haoze Wu, Jia Pan, Peng Huang

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

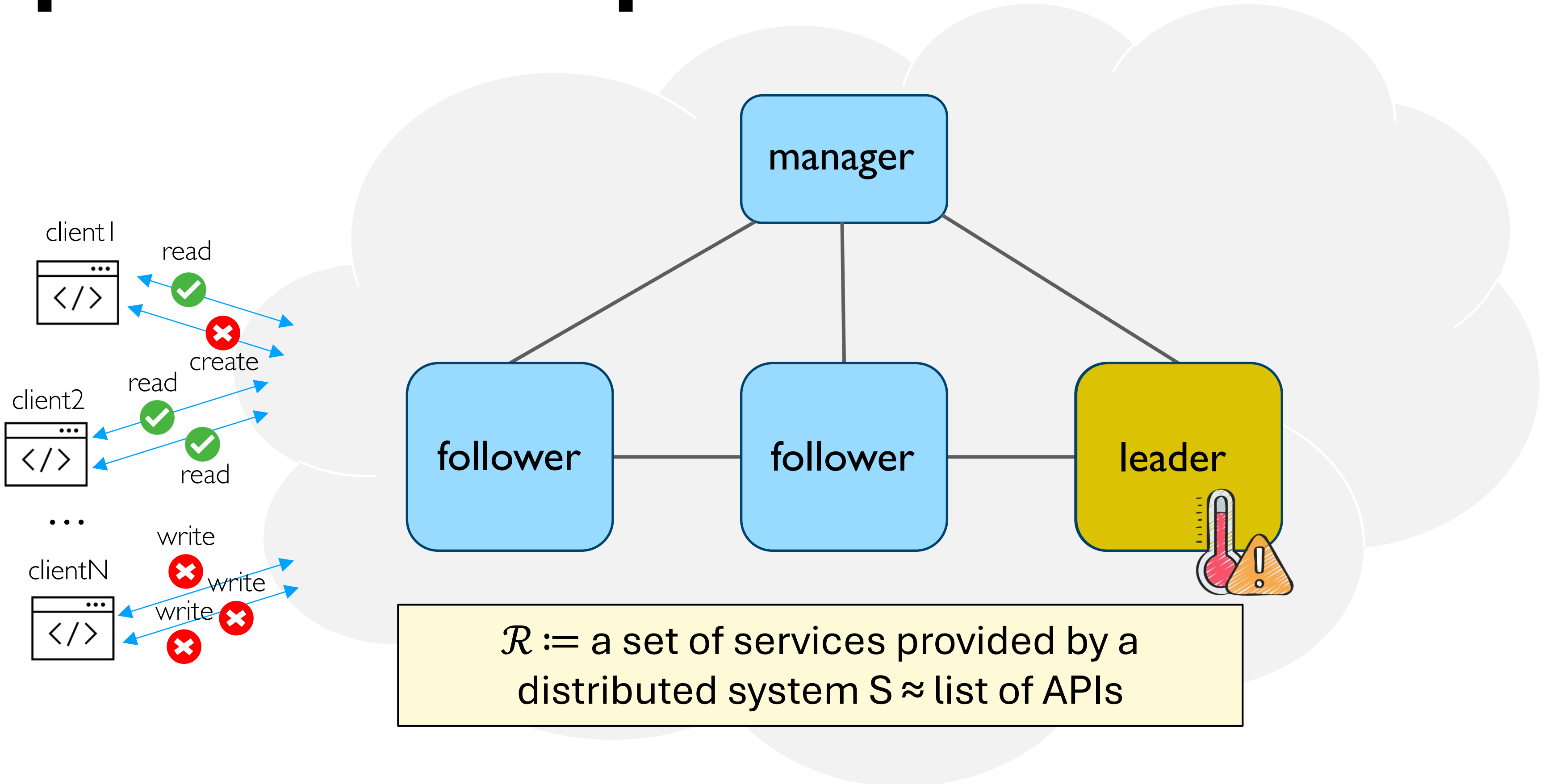
JOHNS HOPKINS UNIVERSITY

NSDI '24

Failures in distributed systems



The problem of partial failures



$\mathcal{R}$ ≔ a set of services provided by a distributed system S ≈ list of APIs

We define partial failure as:

Some $\mathcal{R}_f \subset \mathcal{R}$ fail to maintain their safety or liveness properties, while other services $\mathcal{R} \setminus \mathcal{R}_f$ behave as expected

Partial failures prevalent in production

slack | Status

Friday November 3, 2023

Outage
Users unable to connect to Slack or send messages

Issue summary:
On November 4, 2023, between 5:09 PM PDT and 5:20
were unable send messages or to connect to Slack.

A routine code change introduced a database erro

AWS US East region endures eight-hour wobble thanks to 'Stuck IO' in Elastic Block Store

EC2 instances were impaired, Redshift hurt, and some of you may still struggle to access your data

16

Simon Sharwood, APAC Editor   Tue 28 Sep 2021 // 07:57 UTC

Amazon Web Services' largest region yesterday experienced an eight-hour disruption with the Elastic Block Store (EBS) service that impacted several notable web sites and services.

The lack of fun started at 8:11pm PDT on Sunday, when EBS experienced "degraded

Google Cloud Infrastructure Components Incident #20013

Google Cloud services are experiencing issues and we have an other update at 5:30 PDT

Incident began at **2020-12-14 04:07** and ended at **2020-12-14 06:23** (all times are **US/Pa**

DATE	TIME	DESCRIPTION
Dec 22, 2020	16:49	The following is a correction to the previously posted IS amendment. All services that require sign-in via a Goog Cloud service accounts experienced elevated error rate oauth2.googleapis.com. Impact varied based on the Clo impacted and have further questions.
Dec 18, 2020	11:37	ISSUE SUMMARY On Monday 14 December, 2020, for a duration of 47 mi access were unavailable. Cloud Service accounts used b apologize to our customers whose services or business to improve the platform's performance and availability.

Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region

December 10th, 2021

We want to provide you with some additional information about the service disruption that occurred in the Northern Virginia (US-EAST-1) Region on December 7th, 2021.

Issue Summary

To explain this event, we need to share a little about the internals of the AWS network. While the majority of AWS services and all customer applications run within the main AWS network, AWS makes use of an internal network to host foundational services including monitoring, internal DNS, authorization services, and parts of the EC2 control plane. Because of the importance of these services in this internal network, we connect this network with multiple geographically isolated networking devices and scale the capacity of this network significantly to ensure high availability of this network connection. These networking devices provide additional routing and network address translation that allow AWS services to communicate between the internal network and the main AWS network. At 7:30 AM PST, an automated activity to scale capacity of one of the AWS services hosted in the main AWS network triggered an unexpected behavior from a large number of clients inside the internal network. This resulted in a large surge of connection activity that overwhelmed the networking devices between the internal network and the main AWS network, resulting in delays for communication between these networks. These delays increased latency and errors for services communicating between these networks, resulting in even more connection attempts and retries. This led to persistent congestion and performance issues on the devices connecting the two networks.

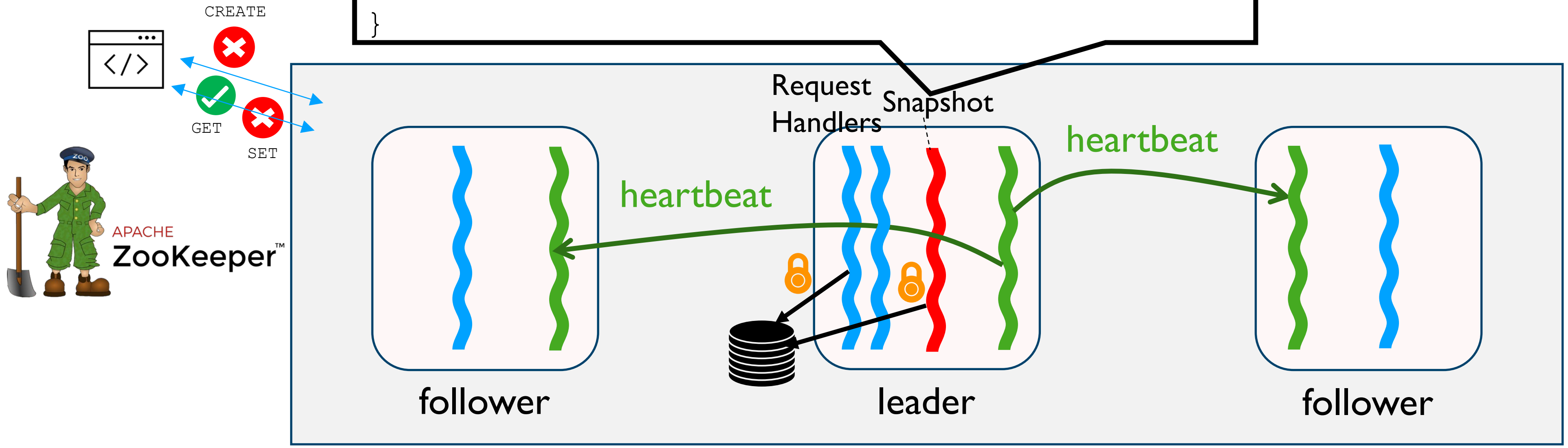
…

A real bug example

```
void serializeNode(OutputArchive oa, ...){
  synchronized (node) {
    scount++;
    oa.writeString(pathString, "path");
    oa.writeRecord(node, "node");
    ...
  }
}
```

Stuck due to transient network issue

Fix: make a copy of node, serialize data without lock



No leader re-election triggered!

Fault injection testing is needed

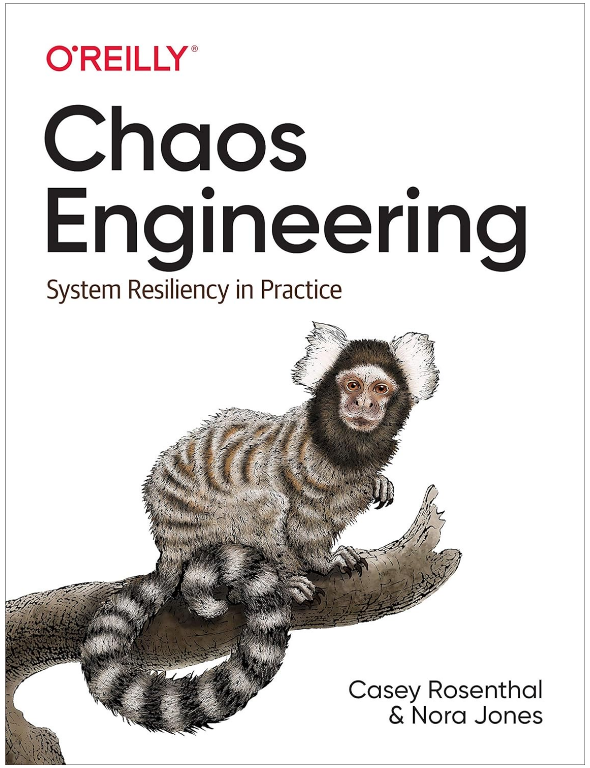
Many partial failure bugs are only triggered by rare fault events

Fault injection testing aims to catch such bugs

- Simulate faults while exercising a system with test workloads

An increasingly popular practice

- Randomly kill a process/VM, introduce packet loss, simulate disk errors, etc.



Chaos Monkey

Jepsen

Blockade

Challenges

```
synchronized (node) {
  oa.writeString(pathString, "path");
  oa.writeRecord(node, "node");
  ...
}
```

network delay should occur to only these operations

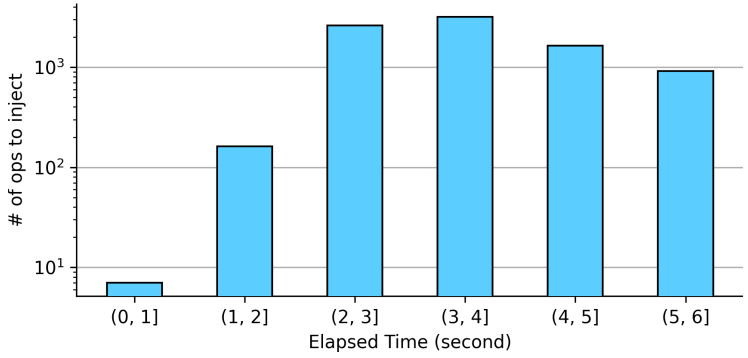
Subtle faults occurring at fine granularity

- only writes to certain files fail
- network fault that only affects a particular connection
- microburst or transient slowness in a subset of operations
- a custom exception in a specific RPC

Coarse-grained, black-box fault injections are insufficient!

Challenges (cont'd)

Very large injection space



- Over **1000** candidates for fault injection in **1** second during ZooKeeper's execution

Distributed systems are by design fault-tolerant

- Most injected faults would be masked or lead to expected behavior (e.g., abort on failure to read a critical file)

random injections are ineffective and inefficient

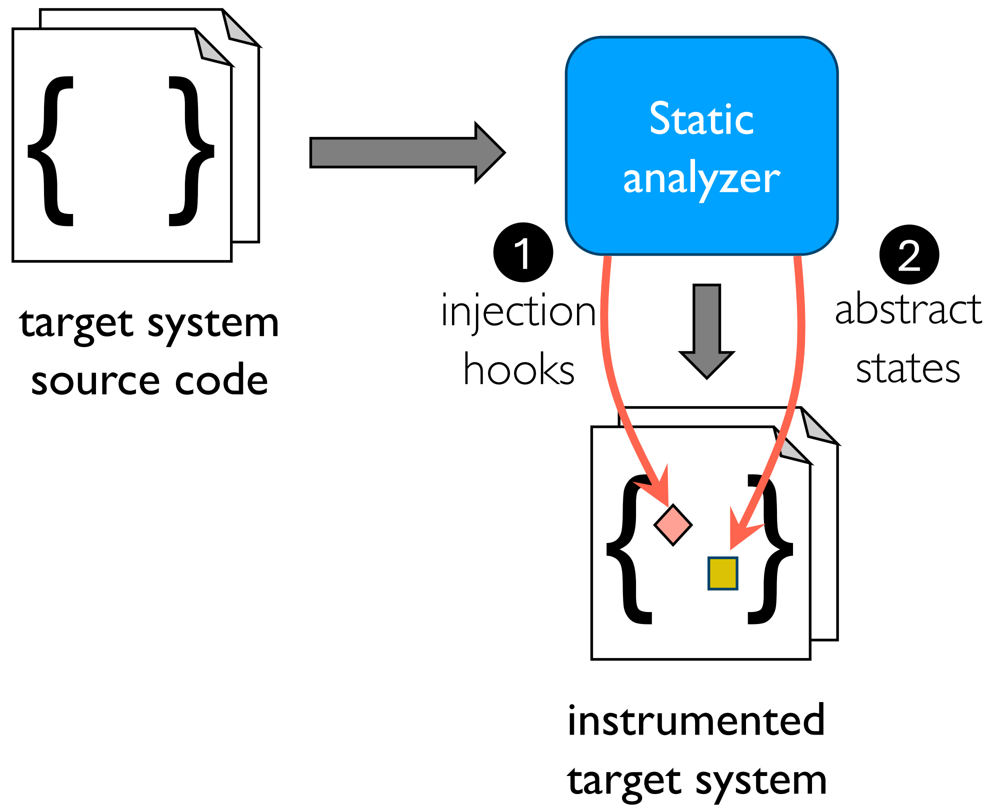
Our solution: Legolas

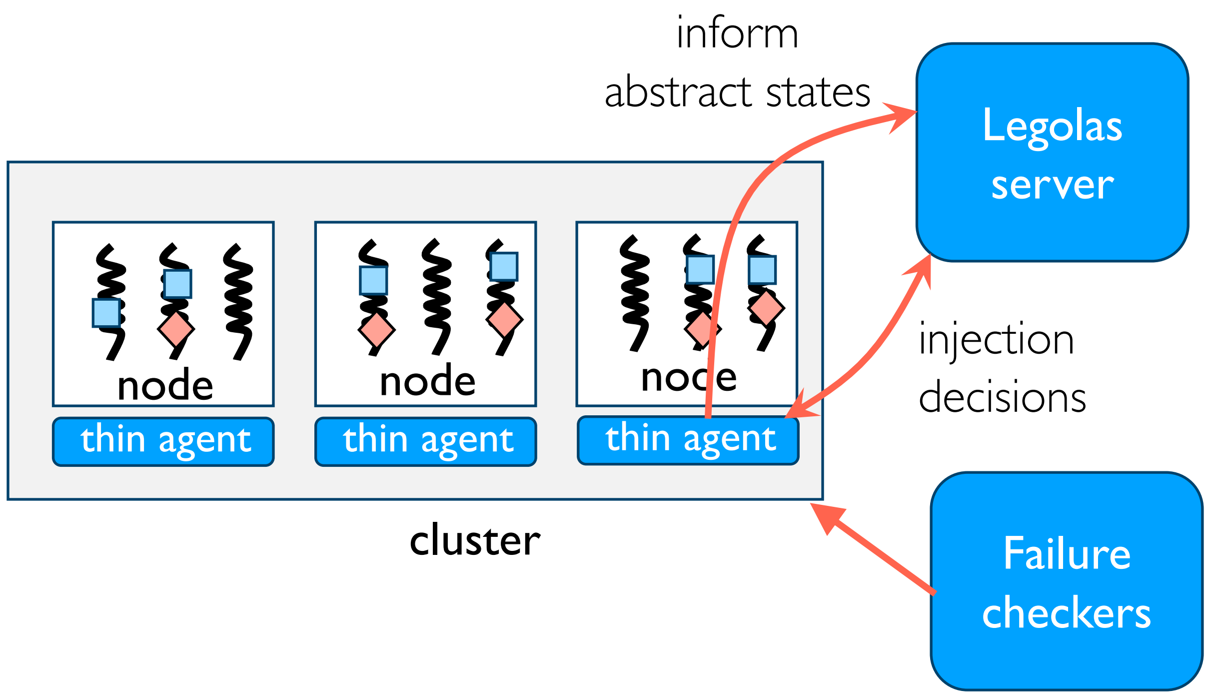
A fault injection testing framework for large distributed system to expose partial failure bugs

1. fine-grained, in-situ injection customized to system code
   - *statically instrument* hooks to precisely simulate subtle faults within a system

2. systematic yet efficient search of large fault injection space
   - extract *abstract state* and leverage the state to compress the search space

Legolas Overview





Analysis stage

Testing stage

Legolas workflow

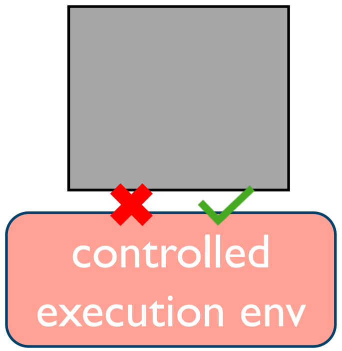
1. **Instrument fault injection hooks**
2. **Extract abstract states**
3. **Stateful injection decision algorithms**
4. **Failure checkers**

This talk

Fault injection granularity & methodology

Node level

process kill,
packet loss,
disk faults, etc.

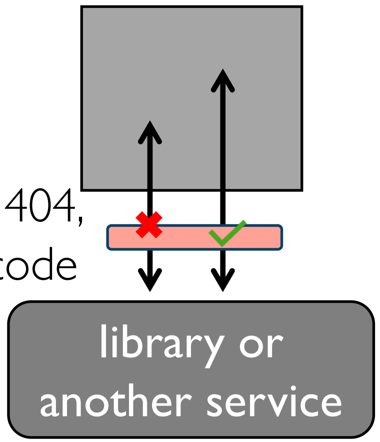


Environment injection

E.g., ChaosMonkey, Blockade, ...

API level

return 404,
error code

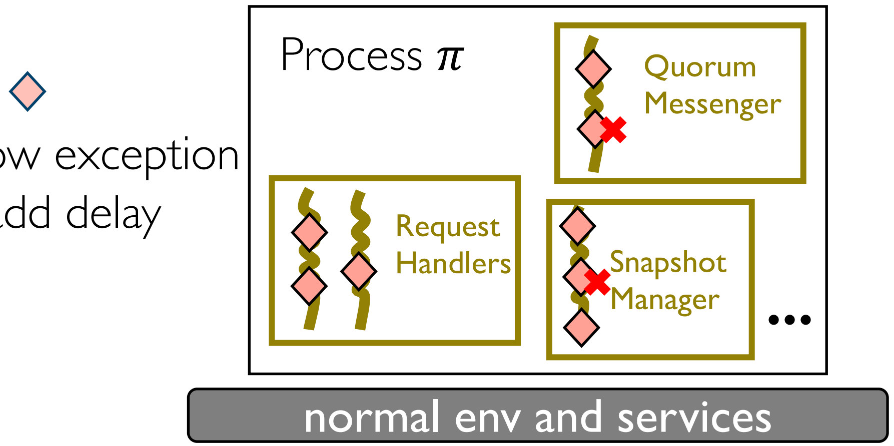


Interception injection

E.g., LFI [DSN '09], Filibuster [SoCC '21]

Operation level

throw exception
or add delay



Instrumentation injection

Legolas

Identify potential faulty conditions

Locate all the call instructions in the code

Analyze the invocation target to extract potential errors

- Based on exceptions in method signatures? Not reliable!

Problem 1: a method may internally throw an exception that is not declared in the signature

Solution: intra-procedural analysis of method body

- Identify exceptions that are uncaught or caught but rethrown

Identify potential faulty conditions

Problem 2: a method may be impossible to encounter an exception declared in the signature

- Due to polymorphism or interface

```java
class ZooKeeperServer {
  void finishSessionInit() {
    ByteArrayOutputStream baos =
       new ByteArrayOutputStream();
    BinaryOutputArchive bos = new
       BinaryOutputArchive(baos);
    bos.writeInt(-1, "len");
  }
}
```

in-memory stream

impossible to throw IOException here!

```java
class BinaryOutputArchive {
  DataOutput out;
  BinaryOutputArchive(OutputStream strm) {
    out = new DataOutputStream(strm);
  }
  void writeInt(int i, String tag) throws IOException {
    out.writeInt(i);
  }
}
```

Solution: context-sensitive, inter-procedural analysis

- Check if objects used in a call site are known in-memory object types

Instrument injection hooks

```
InjectionQuery query = new InjectionQuery(
    serverId, threadId, ..., invokedMethodSig,
    faultIds);
InjectionCommand command = stub.inject(query);
if (command.id == -1) return; // no injection
/* simulate the decided fault */
...
```

```
+ LegolasAgent.inject(0, 3,
    "org.apache.zookeeper.server.DataTree",
    "serializeNode",
    1115, "<org.apache.jute.OutputArchive:
    void writeRecord(...)>", 268);

outputArchive.writeRecord(node, "node");
```

target system code (ZooKeeper)

Legolas agent

Legolas server

As deep as possible for ease of reasoning

A call instruction is injected when:

- Faults originate from explicit *throw* inside the target method body
- Invocation target is an external function

Legolas workflow

1. Instrument fault injection hooks
2. **Extract abstract states**
3. Stateful injection decision algorithms
4. Failure checkers

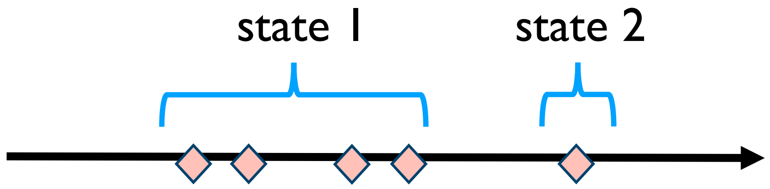
Idea: group injections by execution state

```java
public class SyncRequestProcessor extends Thread {
  public void run() {
    int logCount = 0;
    while (true) {
      Request si = queueRequests.take();
      if (zks.getDB().append(si)) {
        logCount++;
        if (logCount > snapCount) {
          (snapThd = new Thread(() -> {
              zks.takeSnapshot();
          })).start();
        }
        logCount = 0;
      }
    }
    ...
  }
}
```

Most faults
come from here

Few faults com
e from here

```java
synchronized (node) {
  oa.writeString(pathString, "path");
  oa.writeRecord(node, "node");
  ...
}
```

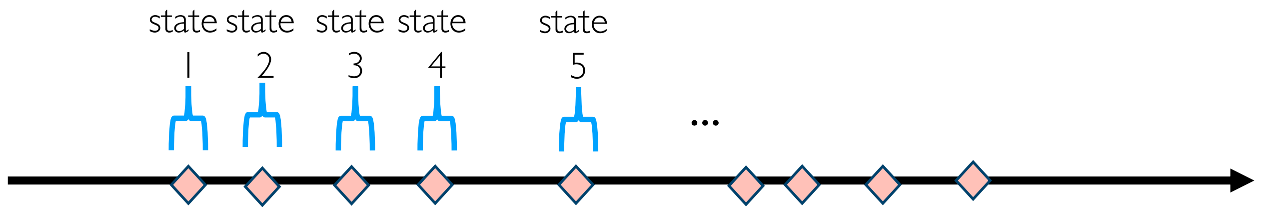


Many fault injection attempts are testing similar scenarios

- Extract high-level state to group fault injection attempts
- Explore injection space systematically with the abstraction of states

State representation

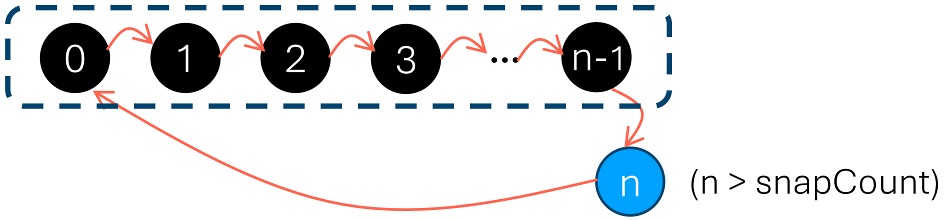
Complete execution state: PC, stacktrace, memory



- Too expensive to track
- Useless grouping

State variables and their value changes?

- Still too excessive → ineffective grouping
- Example: `logCount` as a state variable



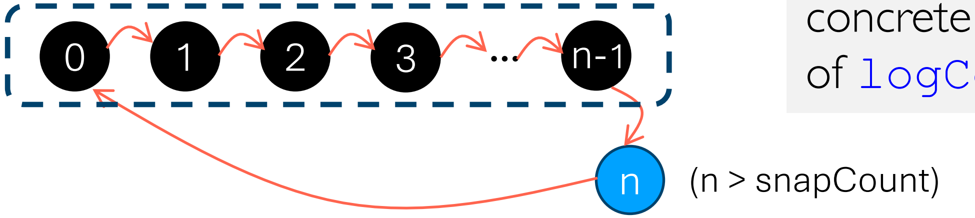

```
int logCount = 0;
while (true) {
  Request si = queuedRequests.take();
  if (zks.getDB().append(si)) {
    logCount++;
    if (logCount > snapCount) {
      zks.takeSnapshot();
      logCount = 0;
    }
  }
  ...
}
```

Need a higher-level representation

Idea: abstract state variables (ASV)

Concrete state values do not matter

…unless they indicate a condition change



concrete values of `logCount`

```
int logCount = 0;
while (true) {
  Request si = queuedRequests.take();
  if (zks.getDB().append(si)) {
    logCount++;
    if (logCount > snapCount) {
      zks.takeSnapshot();
      logCount = 0;
    }
  }
  ...
}
```

- Trigger different code blocks to execute

Each ASV represents a stage of service in the system

Automatically infer abstract states

1. Focus on task-unit classes
   - E.g., classes that extend Thread or Runnable

2. Treat all non-static, non-constant fields in a task class as concrete state variables (SV)

3. Identify the branch conditions that have data-dependency with SVs

4. Locate basic blocks that are control-dependent on these conditions

5. Assigns index for each block as an Abstract State Variable (ASV)

Algorithm 1: Infer abstract state variables

```
Function InferASV(task_class):
    csv_list ← InferCSV(task_class);
    task_method ← getTaskMethod(task_class);
    dep_graph ← buildDependence(task_method, csv_list);
    asv_locations ← [task_method.body().getFirst()];
    Process(task_method.body(), dep_graph, false);
Function Process(instructions, dep_graph, flag):
    inst ← instructions.begin();
    hasAction ← false;
    while inst ≠ instructions.end() do
        if isBranch(inst) then
            <cond, blocks, next> ← parseBranch(inst);
            if dep_graph.contains(cond) then
                for block ← blocks do
                    Process(block.body(), dep_graph, true);
                end
            end
            inst ← next;
        else
            hasAction ← hasAction | isAction(inst);
            inst ← inst.next();
        end
    end
    if hasAction and flag then
        asv_locations.add(instructions.begin());
```

See paper for more details!

Example of ASV inference

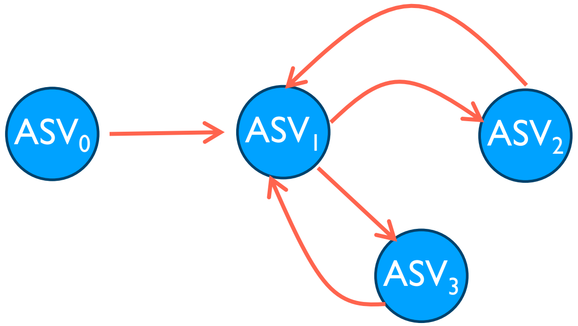
```
public class SyncRequestProcessor extends Thread {
  public void run() {
    int logCount = 0;
+   LegolasAgent.inform(identityHashCode, ..., 0);
    while (true) {
      Request request = queuedRequests.take();
      if (request == requestOfDeath) break;
+     LegolasAgent.inform(identityHashCode, ..., 1);
      if (zks.getZKDatabase().append(request)) {
        if (++logCount > snapCount) {
          if (snapThd != null && snapThd.isAlive()) {
+           LegolasAgent.inform(identityHashCode, ..., 2);
            LOG.warn("Too busy to snap, skipping");
          }
          else {
+            LegolasAgent.inform(identityHashCode, ..., 3);
           (snapThd = new Thread("Snapshot") {...}).start();
          }
          logCount = 0;
        }
      }
    }
  }
}
```

ASV_0 (init)

ASV_1 (request handling)

ASV_2 (snapshotting in progress)

ASV_3 (snapshotting)

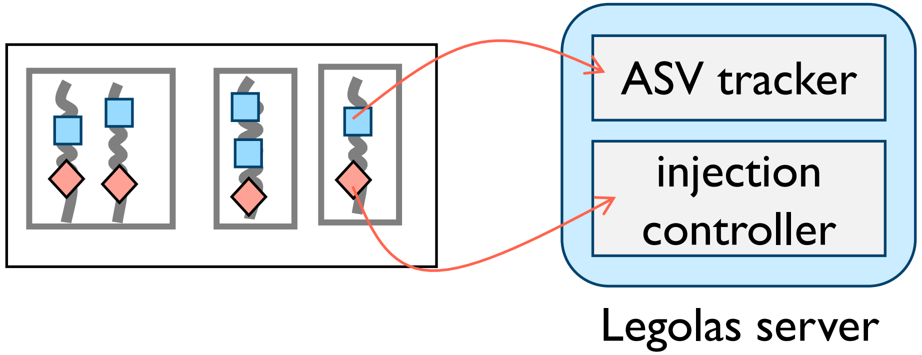


Legolas workflow

1. Instrument fault injection hooks
2. Extract abstract states
3. **Stateful injection decision algorithms**
4. Failure checkers

Injection decision algorithm

Use the current ASVs to decide whether to grant an injection or not

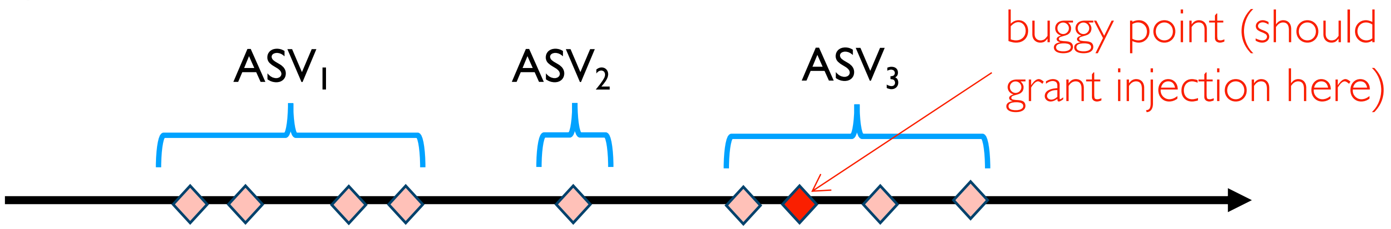


Consideration 1: Should not focus too much on one state

- Difficult to know if a state is interesting or not

Consideration 2: Buggy point may not be the first request

- Trying just once can miss bugs

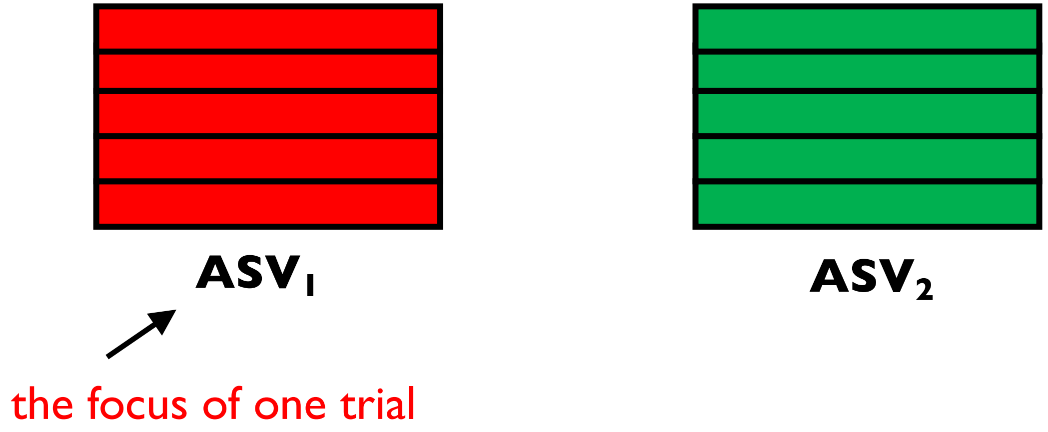


Idea: budgeted-state round-robin (BSRR)

Initial budget for all ASVs is *N* (default 5)

Each trial focuses on one ASV

- Only an injection from this ASV would be granted
- Injections from other ASVs would be denied

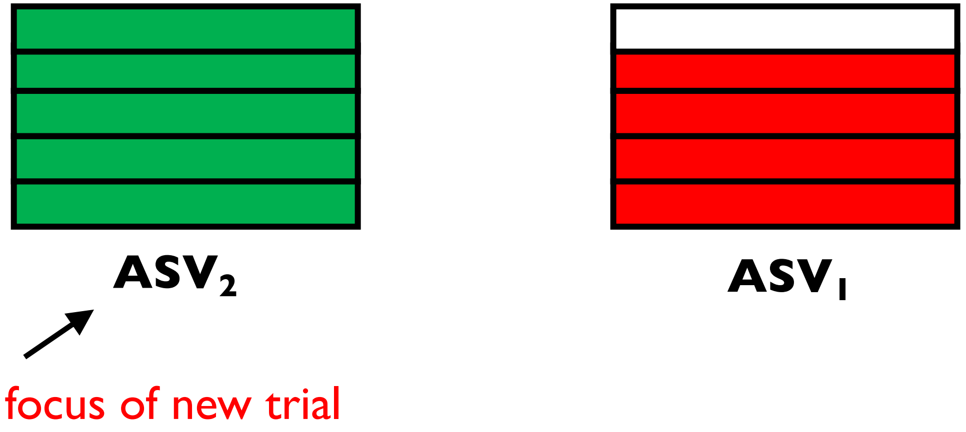


Budgeted-state round-robin (BSRR)

If an injection is granted, decrease the budget by 1

Move focused ASV to the queue end

Focus on ASV at the queue front in next trial

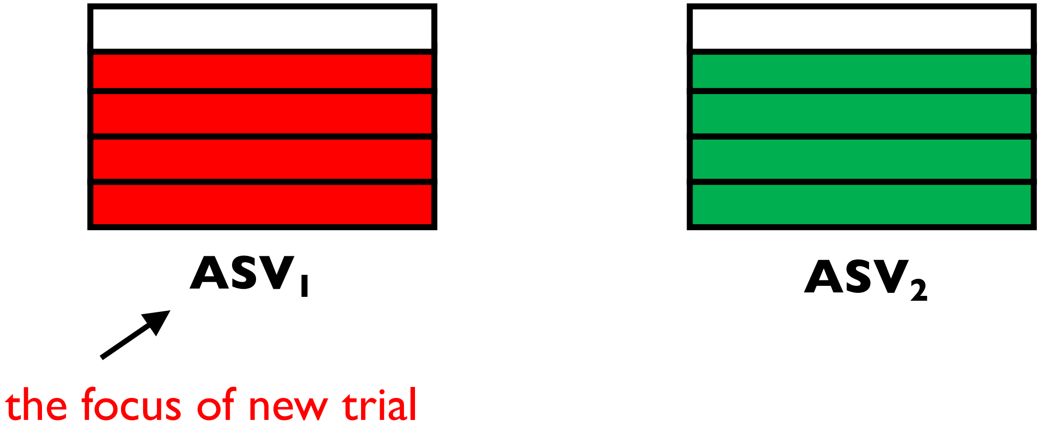


Budgeted-state round-robin (BSRR)

If an injection is granted, decrease the budget by 1

Move focused ASV to the queue end

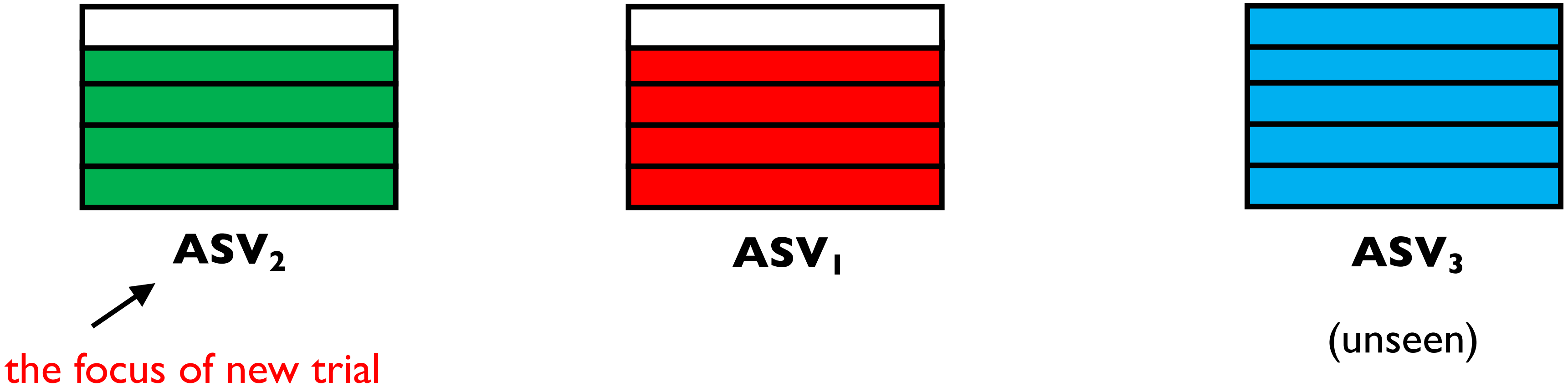
Focus on ASV at the queue front in next trial



Budgeted-state round-robin (BSRR)

If an ASV is unseen before, append it to the queue end

If an ASV's budget is used up, skip it in the round-robin

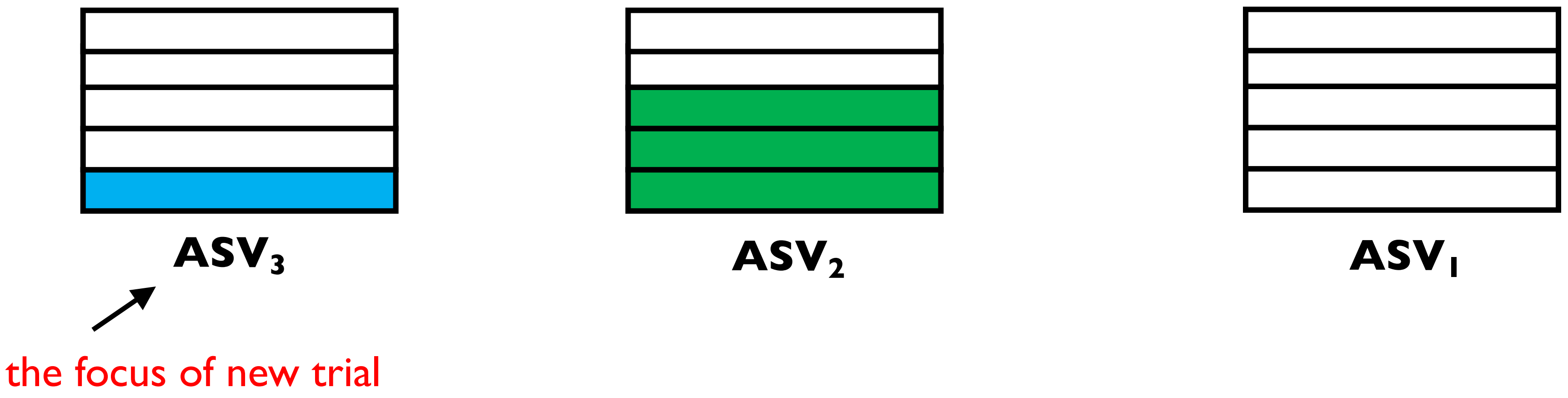


Budgeted-state round-robin (BSRR)

If an ASV is unseen before, append it to the queue end

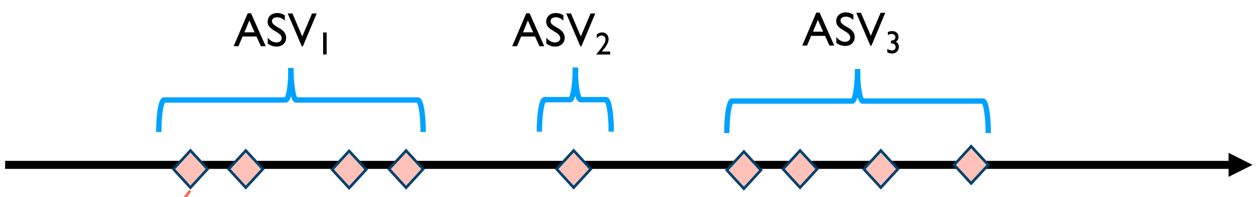
If an ASV's budget is used up, skip it in the round-robin

After all ASVs' budgets are used up, refill all ASVs



Randomization within an ASV

If there are multiple injection requests in an ASV, use randomization



probability an injection request is granted

$$p = 1 - e^{\ln 0.01 / c}$$

# of injection requests in an ASV

Rationale:

- Grant at least one request from this ASV
- Let injection occur neither too early nor too late

Experiment setup

Evaluated systems

- Six widely-used, large-scale distributed systems

System	Release	SLOC	Type
ZooKeeper	3.6.2	95K	Coordination service
HDFS	3.2.2	689K	Distributed file system
Kafka	2.8.0	322K	Event streaming system
HBase	2.4.2	728K	Distributed database
Cassandra	3.11.10	210K	Distributed database
Flink	1.14.0	78K	Stateful streaming system

Two fault injection testing experiments for each system

1. Exception: (1) I/O related exceptions; (2) custom exception that extends `IOException`
2. Delay: function calls that involve disk or network I/O

Injection instrumentation & ASV extraction

System	Statically injected		Class	ASM	ASV			
	Methods	Points			Total	Mean	Min	Max
ZooKeeper	484	1947	708	36	226	6	1	31
HDFS	2127	3913	4636	104	390	4	1	16
Kafka	343	754	5829	51	220	4	1	15
HBase	5874	11051	10462	96	312	3	1	17
Cassandra	2127	3913	4636	104	390	4	1	18
Flink	997	2299	4852	48	110	2	1	6

ASMs are the task unit classes (e.g., Threads, Runnables, etc)

New bugs found by Legolas

System	Unique Bugs
ZooKeeper	4
HDFS	5
Kafka	5
HBase	2
Cassandra	2
Flink	2

All cause partial failure symptoms

Root causes are diverse

- Logic bugs, design flaws, mishandling of exceptions, race conditions

Eleven reports are explicitly confirmed by developers

New bug example in HDFS

```
class BlockReceiver implements Closeable {
  private int receivePacket() throws IOException {
    ...
    boolean lastPacketInBlock =
        header.isLastPacketInBlock();
    if (mirrorOut != null && !mirrorError) {
      try {
        ...
        packetReceiver.mirrorPacketTo(mirrorOut);
        ...
      } catch (IOException e) {
        handleMirrorOutError(e);
      }
    }
    return lastPacketInBlock?-1:len;
  }
}
```

IOException injected inside

set flag mirrorError

```
class PacketResponder implements Runnable, Closeable {
  public void run() {
    while (isRunning() && !lastPacketInBlock) {
      PipelineAck ack = new PipelineAck();
      try {
        if (type != PacketResponderType.LAST_IN_PIPELINE
              && !mirrorError) {
          ack.readFields(downstreamIn);
        }
        // ...
      } catch (IOException ioe) {
        ...
      }
    }
  }
}
```

get stuck

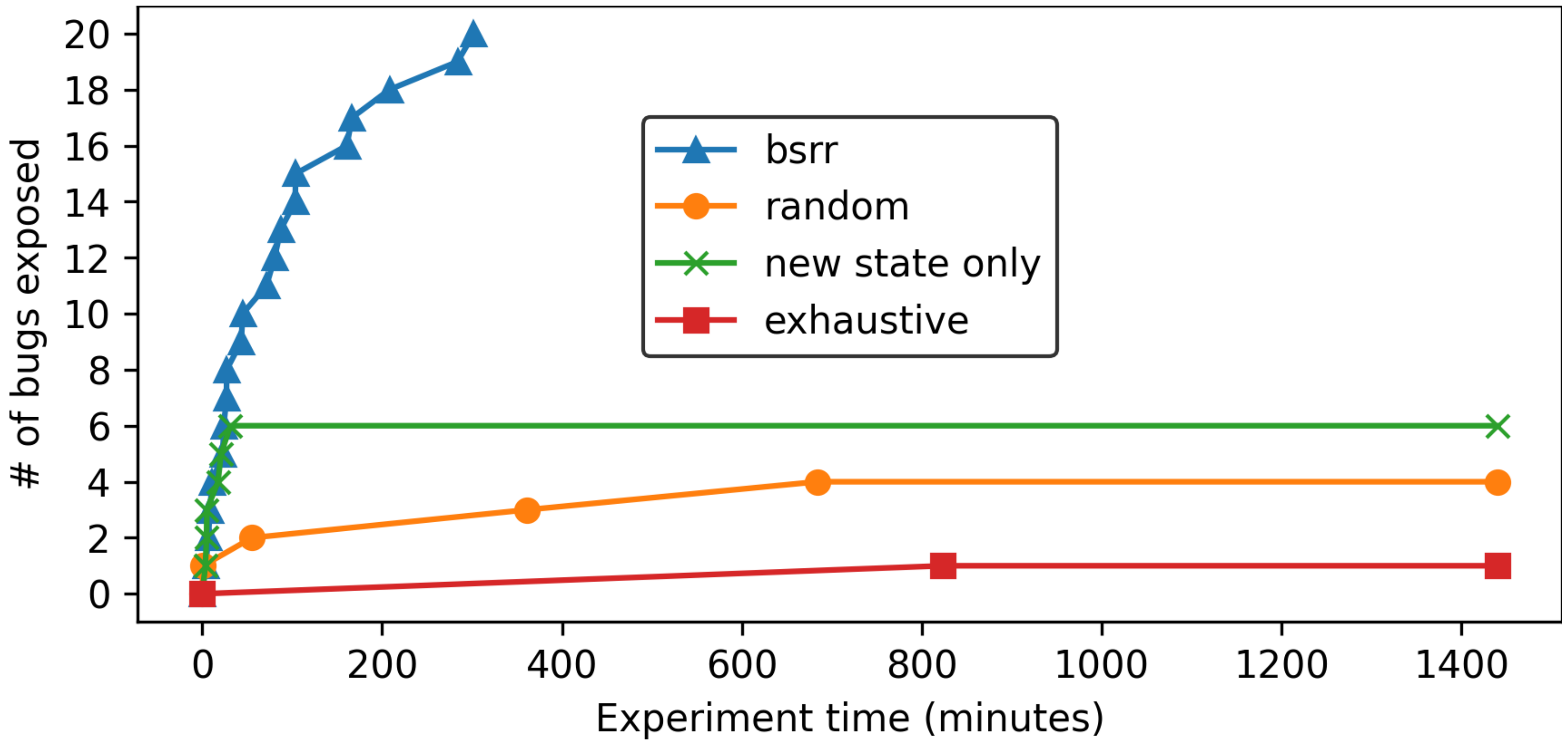
Symptom:

- Some client hangs for 1 minute (normally the client is immediately notified of the error)

Root cause

- The flag `mirrorError` is set after `PacketResponder` checks it

Efficacy of decision algorithm BSRR



BSRR exposed 20 bugs in a median of 58.2 minutes and a minimum of 4 minutes

Comparisons with related work

Work	Description	Exposed bugs	Median detection time
FATE [NSDI '11]	Use a concept of failure IDs to enumerate failures	1	1057.9 minutes
CrashTuner [SOSP '19]	Use meta-info variable accesses to decide the timing of injecting faults	4	20.4 minutes
CORDS [FAST '17]	Use a FUSE file system to inject a single corruption or read/write error to one file-system block at a time	0	N/A

Conclusion

Partial failure bugs are notorious in distributed systems

- Often only occur under subtle faulty conditions at special timing
- Existing fault injection testing is insufficient

Legolas: fault injection testing framework to expose partial failure bugs

1. Perform fine-grained, in-situ injection w/ static instrumentation
2. Automatically extract Abstract State Variables (ASVs) from system code
3. Use ASVs to fault injection decisions

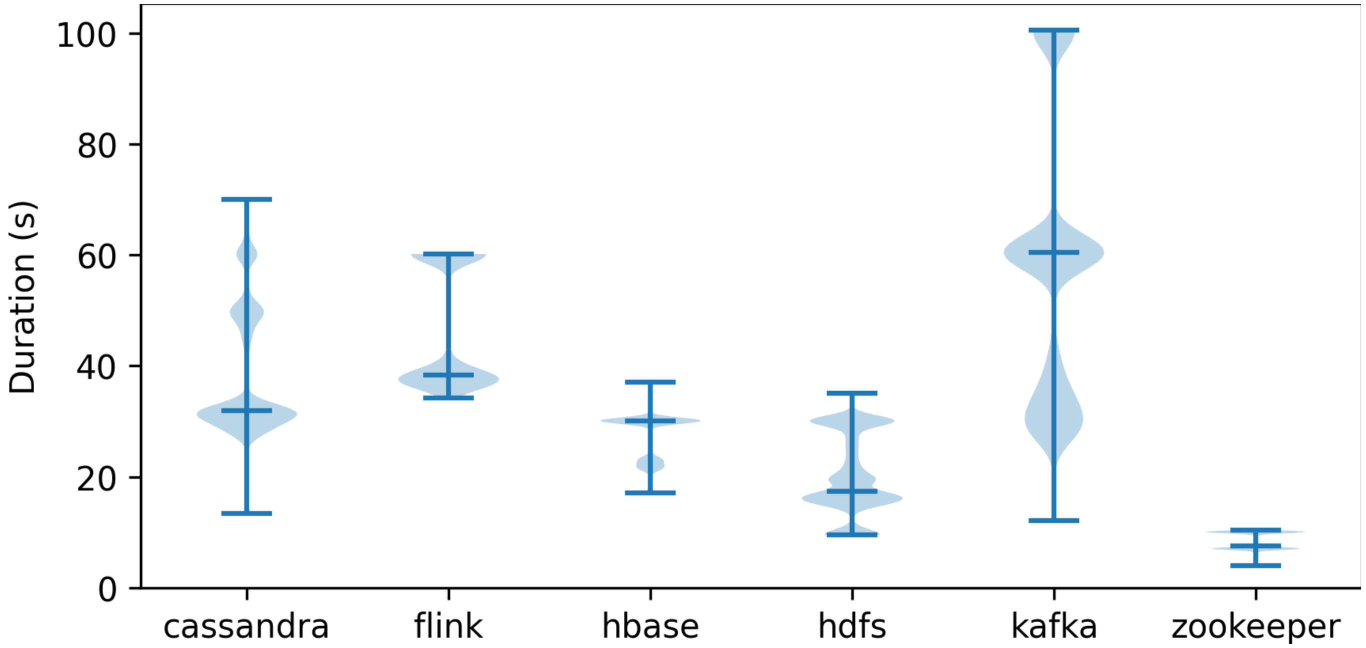
https://github.com/OrderLab/Legolas

Backup slides

Performance of analysis and instrumentation

ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
8.9 sec	31.6 sec	36.9 sec	20.9 sec	77.6 sec	63.9 sec

Fault injection trial duration



Invalid injections

Trials with invalid injections

ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
45 (6)	20 (9)	0	894 (10)	86 (10)	0

All eliminated with context-sensitive invalid injection analysis

Known bugs

System	Bug Id	Exposure Time
ZooKeeper	ZK-2029	15.4 min
	ZK-2201	30.6 min
	ZK-2247	52.1 min
	ZK-2325	2.6 min
	ZK-2982	18.5 min
Cassandra	CA-6364	10.0 min
	CA-6415	330.6 min
	CA-8485	25.3 min
	CA-13833	86.6 min
HDFS	HDFS-11608	29.2 min
	HDFS-12157	39.9 min

Related Work

Partial Failures

- Fail-Stutter [HotOS '01], IRON [SOSP '05], Limplock [SoCC '13], Fail-Slow Hardware [FAST '18], Gray Failure [HotOS '17]

Fault Injection

- FATE [NSDI '11], CrashTuner [SOSP '19], CORDS [FAST '17], CharybdeFS, tcconfig, byte-monkey

Model Checking

- MODIST [NSDI '09], SAMC [OSDI '14], FlyMC [EuroSys '19]