Understanding, Detecting and Localizing Partial Failures in Large System Software

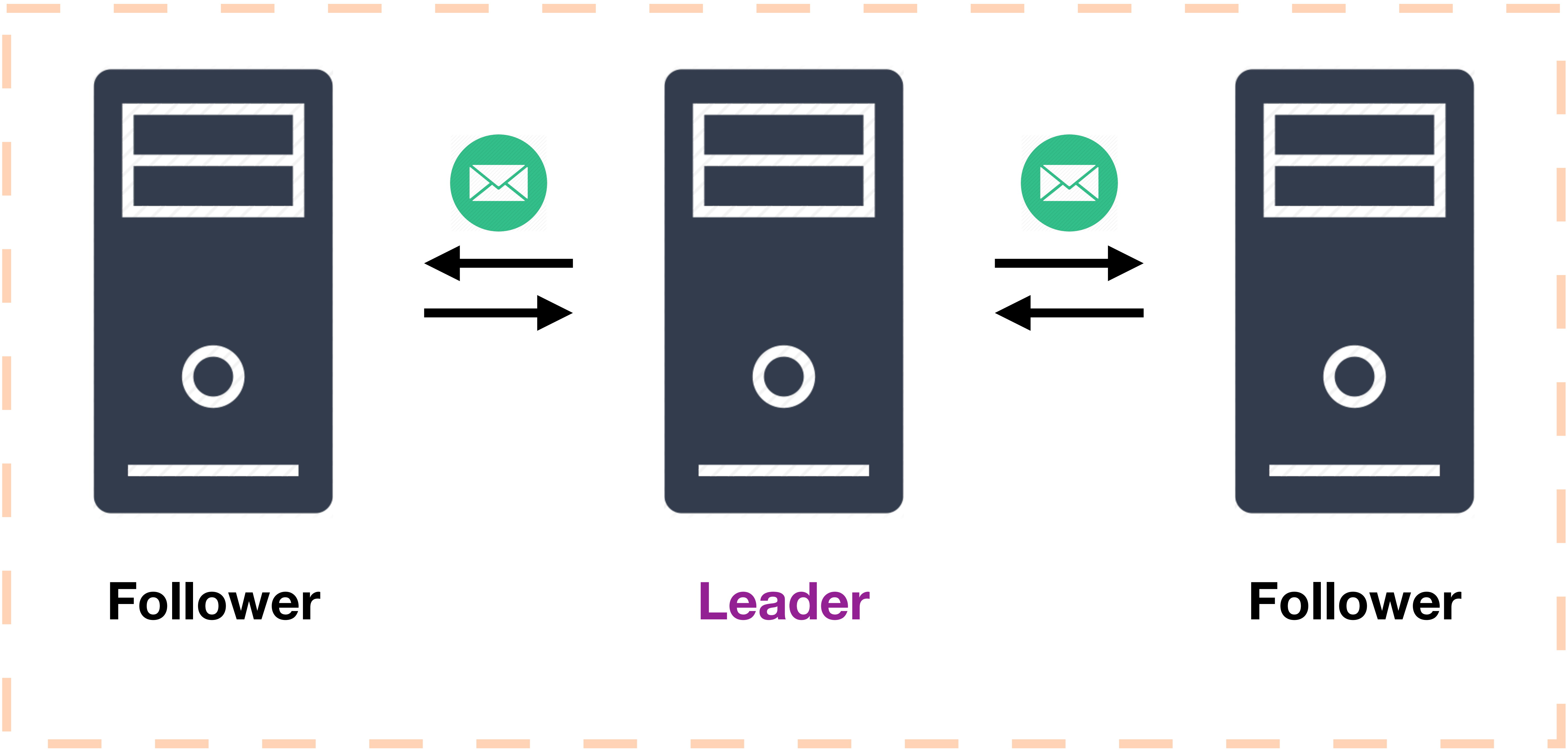
Chang Lou*, Peng Huang, Scott Smith

JOHNS HOPKINS UNIVERSITY

NSDI'20

All start from a puzzling incident..

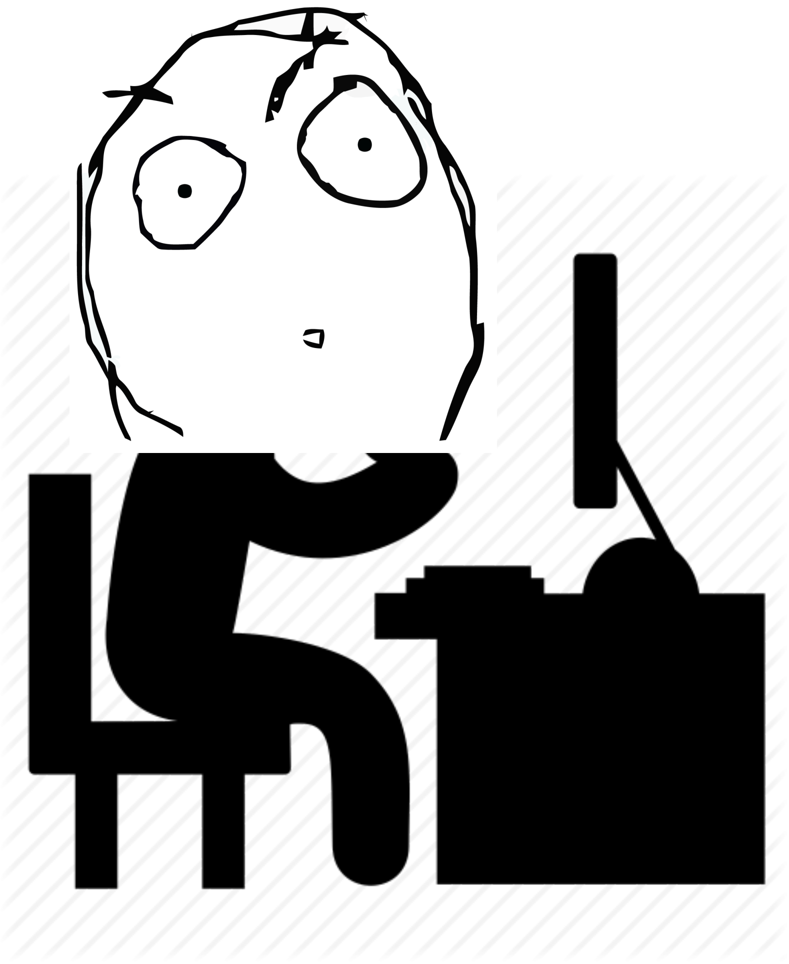
On-call
engineer



ZooKeeper Cluster

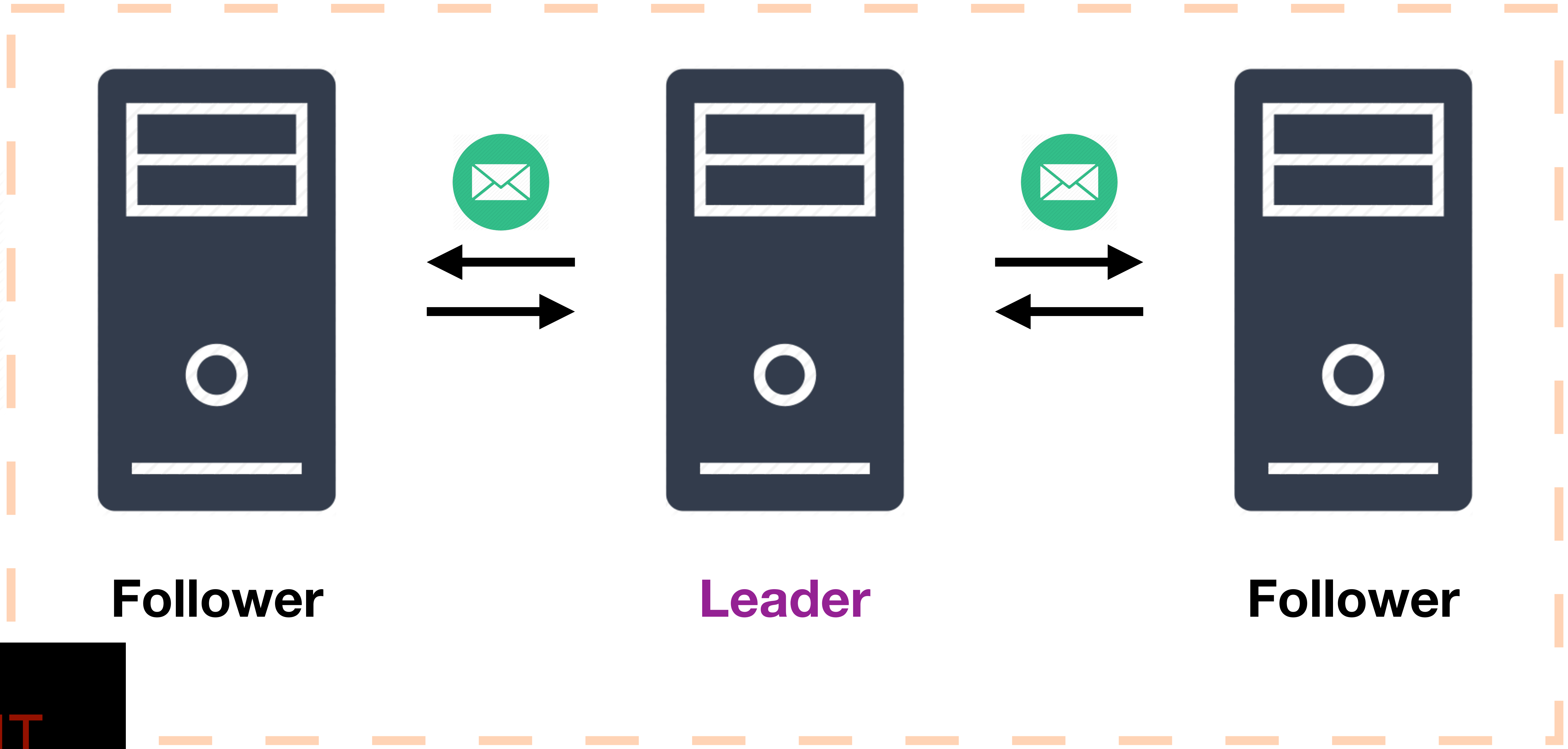
All start from a puzzling incident..

New Message: NEED HELP!
————
We met something puzzling. It has been going on for 10+ minutes.



On-call
engineer

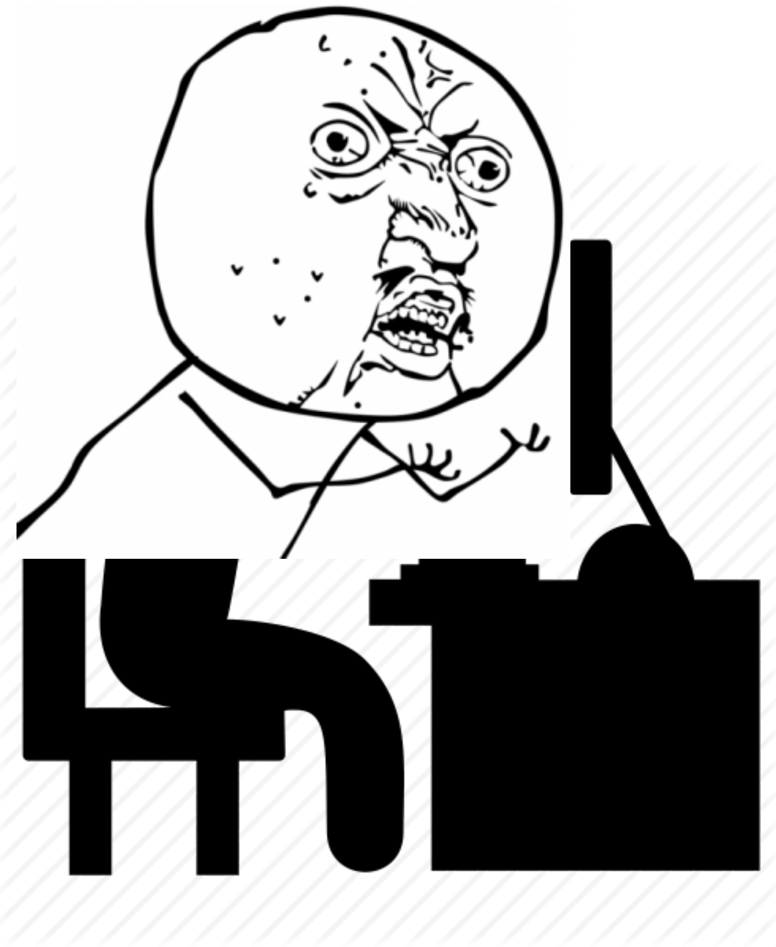
WRITE TIMEOUT
WRITE TIMEOUT
WRITE TIMEOUT



ZooKeeper Cluster

All start from a puzzling incident..

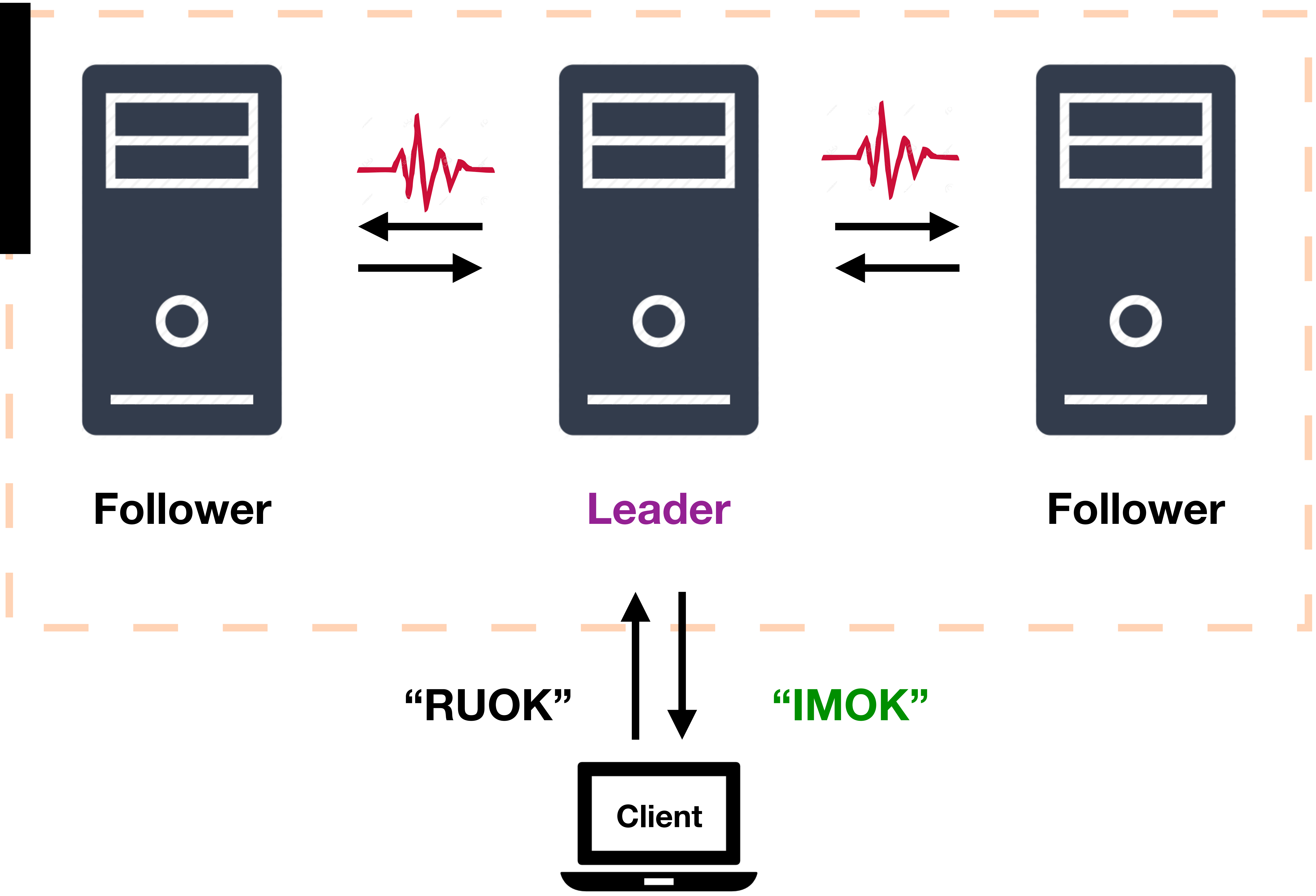
```
READ OK
WRITE TIMEOUT
CREATE TIMEOUT
```



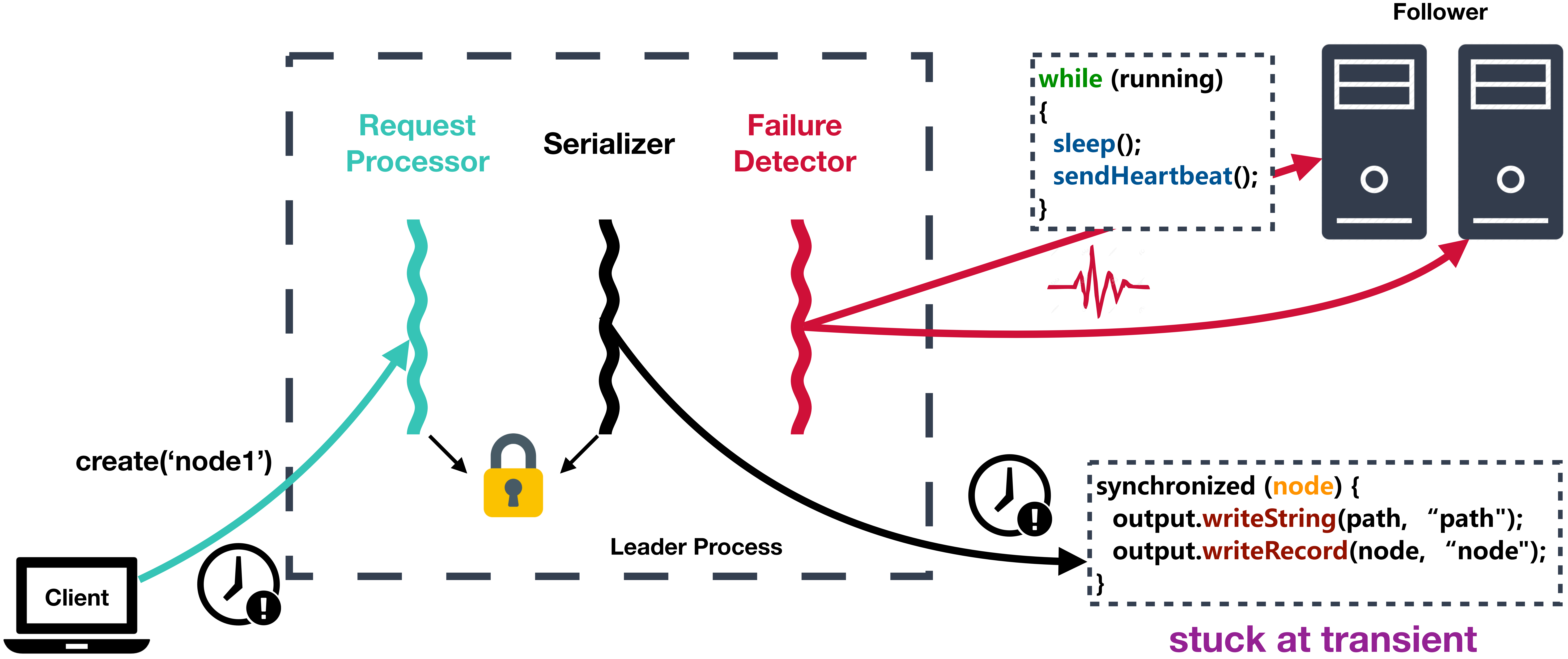
On-call engineer

logs

resource meter



The failure off the radar

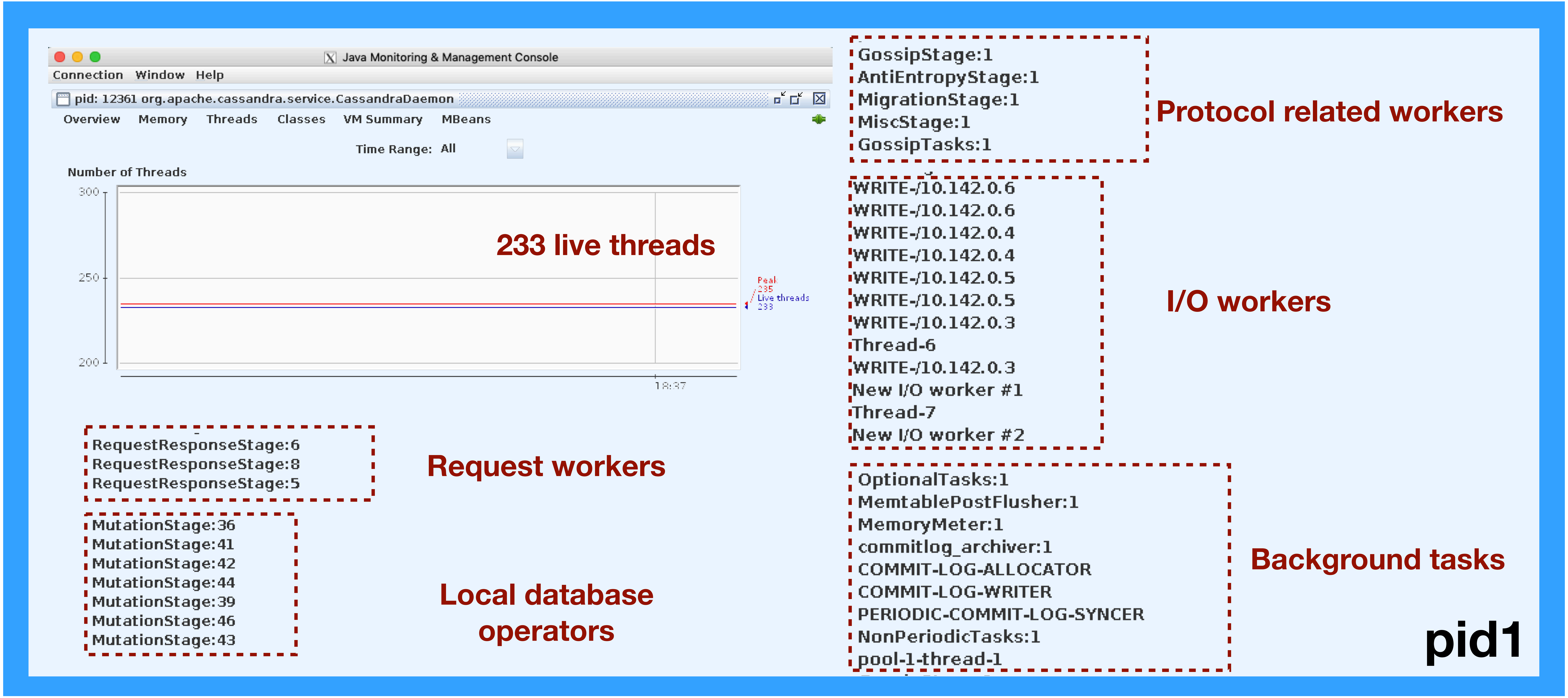


```
while (running)
{
  sleep();
  sendHeartbeat();
}
```

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

stuck at transient network error

Modern software is complex



pid: 12361 org.apache.cassandra.service.CassandraDaemon

233 live threads

```
GossipStage:1
AntiEntropyStage:1
MigrationStage:1
MiscStage:1
GossipTasks:1
```

Protocol related workers

```
WRITE-/10.142.0.6
WRITE-/10.142.0.6
WRITE-/10.142.0.4
WRITE-/10.142.0.4
WRITE-/10.142.0.5
WRITE-/10.142.0.5
WRITE-/10.142.0.3
Thread-6
WRITE-/10.142.0.3
New I/O worker #1
Thread-7
New I/O worker #2
```

I/O workers

```
RequestResponseStage:6
RequestResponseStage:8
RequestResponseStage:5
```

Request workers

```
MutationStage:36
MutationStage:41
MutationStage:42
MutationStage:44
MutationStage:39
MutationStage:46
MutationStage:43
```

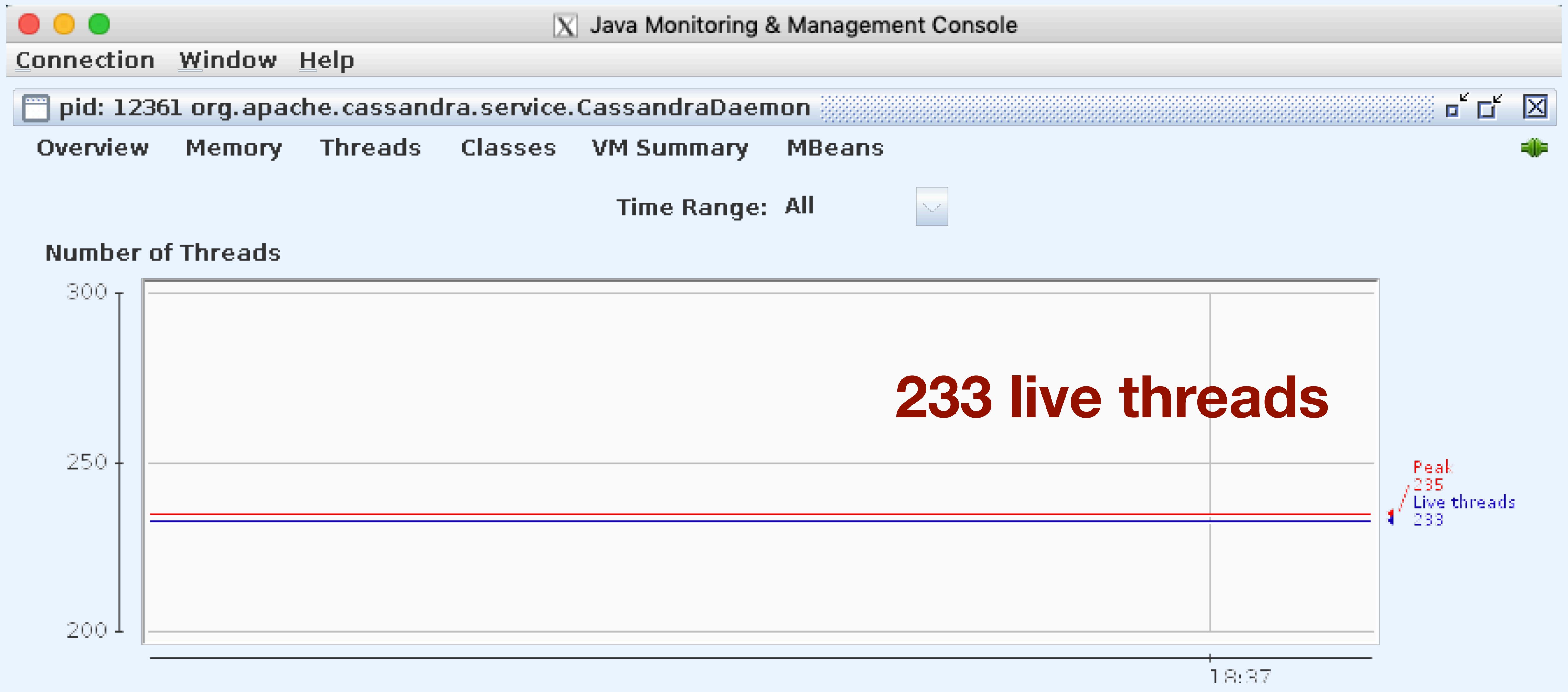
Local database operators

```
OptionalTasks:1
MemtablePostFlusher:1
MemoryMeter:1
commitlog_archiver:1
COMMIT-LOG-ALLOCATOR
COMMIT-LOG-WRITER
PERIODIC-COMMIT-LOG-SYNCER
NonPeriodicTasks:1
pool-1-thread-1
```

Background tasks

pid1

Modern software is complex



Java Monitoring & Management Console
Connection Window Help
pid: 12361 org.apache.cassandra.service.CassandraDaemon
Overview Memory Threads Classes VM Summary MBeans
Time Range: All
Number of Threads
300
250
200
18:37
Peak 235
Live threads 233

233 live threads

```
RequestResponseStage:6
RequestResponseStage:8
RequestResponseStage:5
```

Request workers

```
MutationStage:36
MutationStage:41
MutationStage:42
MutationStage:44
MutationStage:39
MutationStage:46
MutationStage:43
```

Local database operators

```
GossipStage:1
AntiEntropyStage:1
MigrationStage:1
MiscStage:1
GossipTasks:1
```

Protocol related workers

```
WRITE-/10.142.0.6
WRITE-/10.142.0.6
WRITE-/10.142.0.4
WRITE-/10.142.0.4
WRITE-/10.142.0.5
WRITE-/10.142.0.5
WRITE-/10.142.0.3
Thread-6
WRITE-/10.142.0.3
New I/O worker #1
Thread-7
New I/O worker #2
```

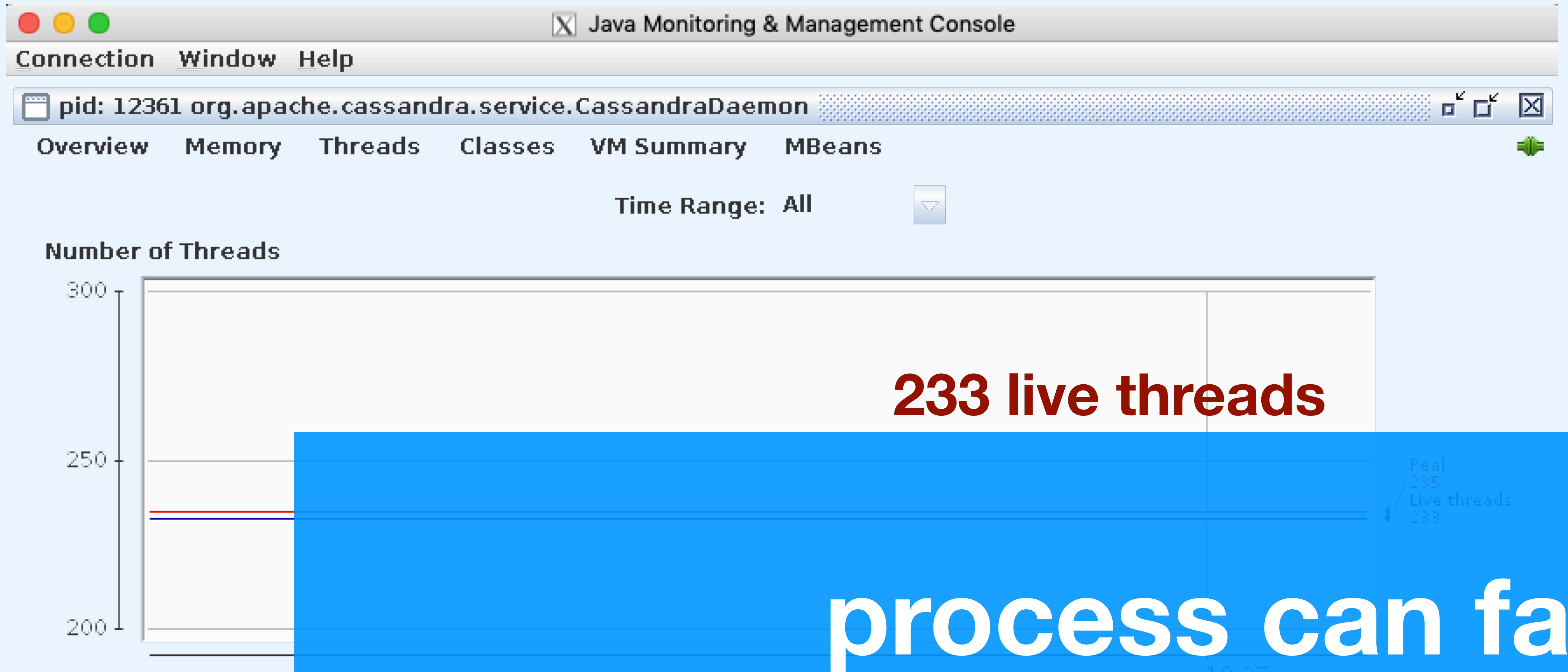
I/O workers

```
OptionalTasks:1
MemtablePostFlusher:1
MemoryMeter:1
commitlog_archiver:1
COMMIT-LOG-ALLOCATOR
COMMIT-LOG-WRITER
PERIODIC-COMMIT-LOG-SYNCER
NonPeriodicTasks:1
pool-1-thread-1
```

Background tasks

pid1

Modern software is complex



pid: 12361 org.apache.cassandra.service.CassandraDaemon

Time Range: All

233 live threads

```
GossipStage:1
AntiEntropyStage:1
MigrationStage:1
MiscStage:1
GossipTasks:1
```

Protocol related workers

```
WRITE-/10.142.0.6
WRITE-/10.142.0.6
WRITE-/10.142.0.4
WRITE-/10.142.0.4
WRITE-/10.142.0.5
WRITE-/10.142.0.5
WRITE-/10.142.0.3
Thread-6
WRITE-/10.142.0.4
New I/O worker #1
Thread-7
New I/O worker #2
```

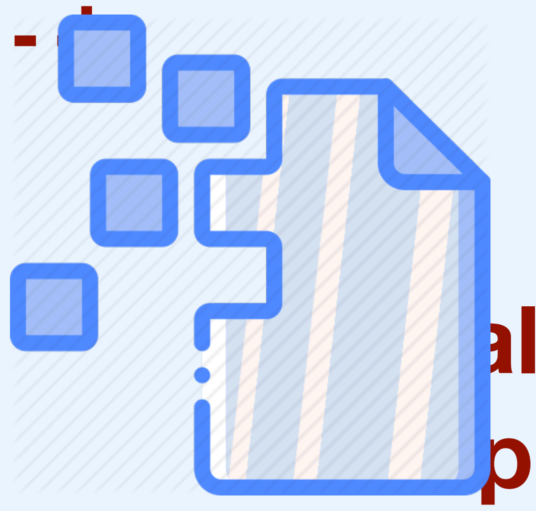
I/O workers

process can fail partially

```
RequestResponseStage:6
RequestResponseStage:8
RequestResponseStage:5
```

Request workers

```
MutationStage:36
MutationStage:41
MutationStage:42
MutationStage:44
MutationStage:39
MutationStage:46
MutationStage:43
```



al database
perators

```
OptionalTasks:1
MemtablePostFlusher:1
MemoryMeter:1
commitlog_archiver:1
COMMIT-LOG-ALLOCATOR
COMMIT-LOG-WRITER
PERIODIC-COMMIT-LOG-SYNCER
NonPeriodicTasks:1
pool-1-thread-1
```

ckground tasks

pid1

Real world outages caused by partial failures

23 May 2013

3 difficult days for Rackspace Cloud Load Balancers

Posted by **iwgcr**

赞 0

The Cloud
many issues

On May 19,
experiencing
04:32 PM ED
but not close

Rackspace C

Load Bala
rare capa

911 emergency services go down across the US after CenturyLink outage

Zack W

Microsoft's MFA is so strong, it locked out users for 8 hours

21 NOV 2018 12

2-factor Authentication, Microsoft, Organisations

Alibaba Cloud Reports IO Hang Error in North China

Th
re
to
go

Amazon Web Services outage once again shows reality behind "the cloud"

reddit, Imgur, and other sites fall offline due to cloud storage failure.

By LEE HUTCHIN

After almost 24 hours of technical difficulties, Facebook is back

Facebook blamed the issue on a "server configuration change."

By Kurt Wagner and Rani Molla | Mar 14, 2019, 1:22pm EDT

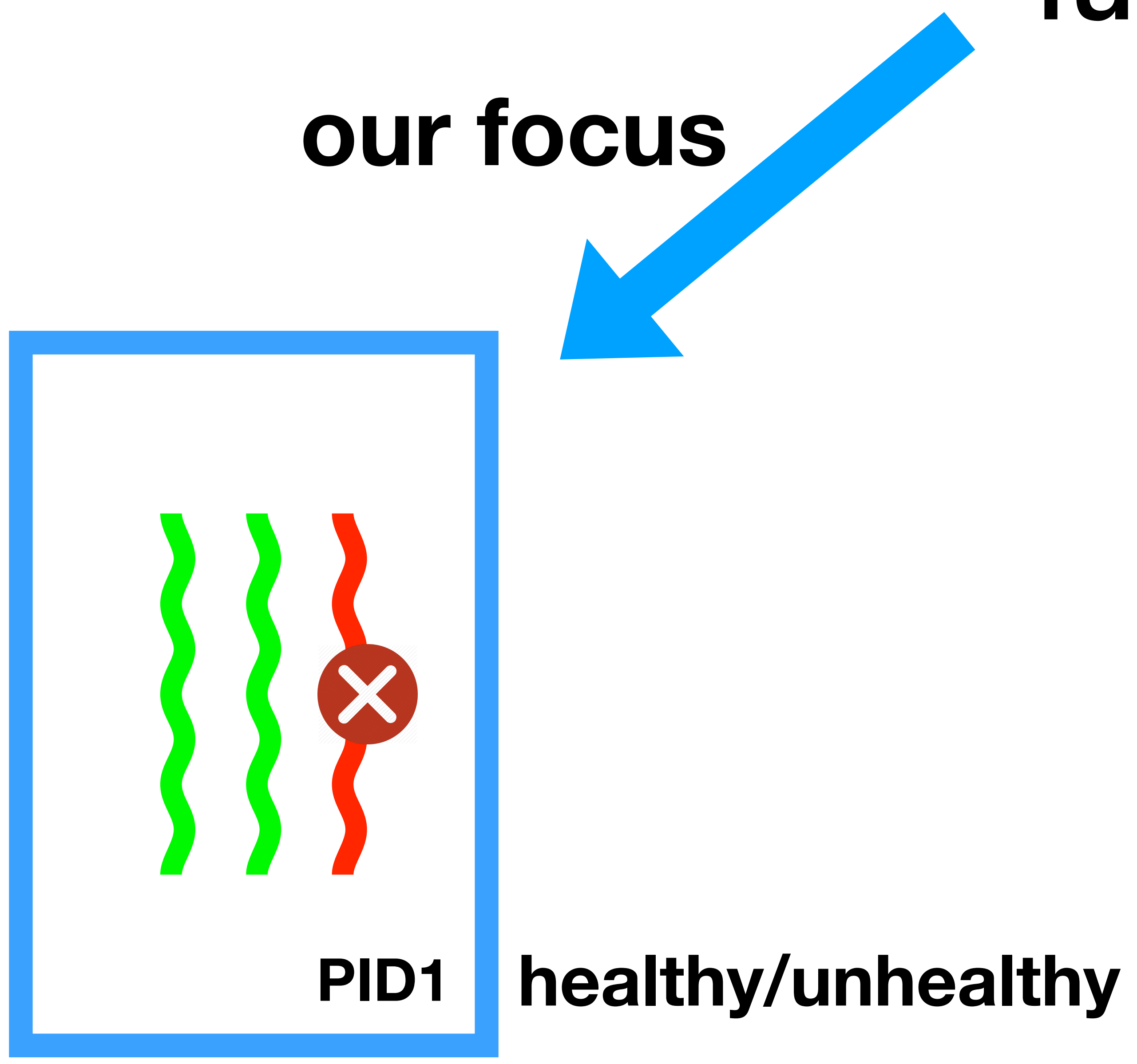
Study methodology

- **We study 100 partial failure cases from five large, widely-used software systems**
  - Interestingly, 54% of them occur in the most recent three years' software releases (average lifespan of all systems is 9 years).

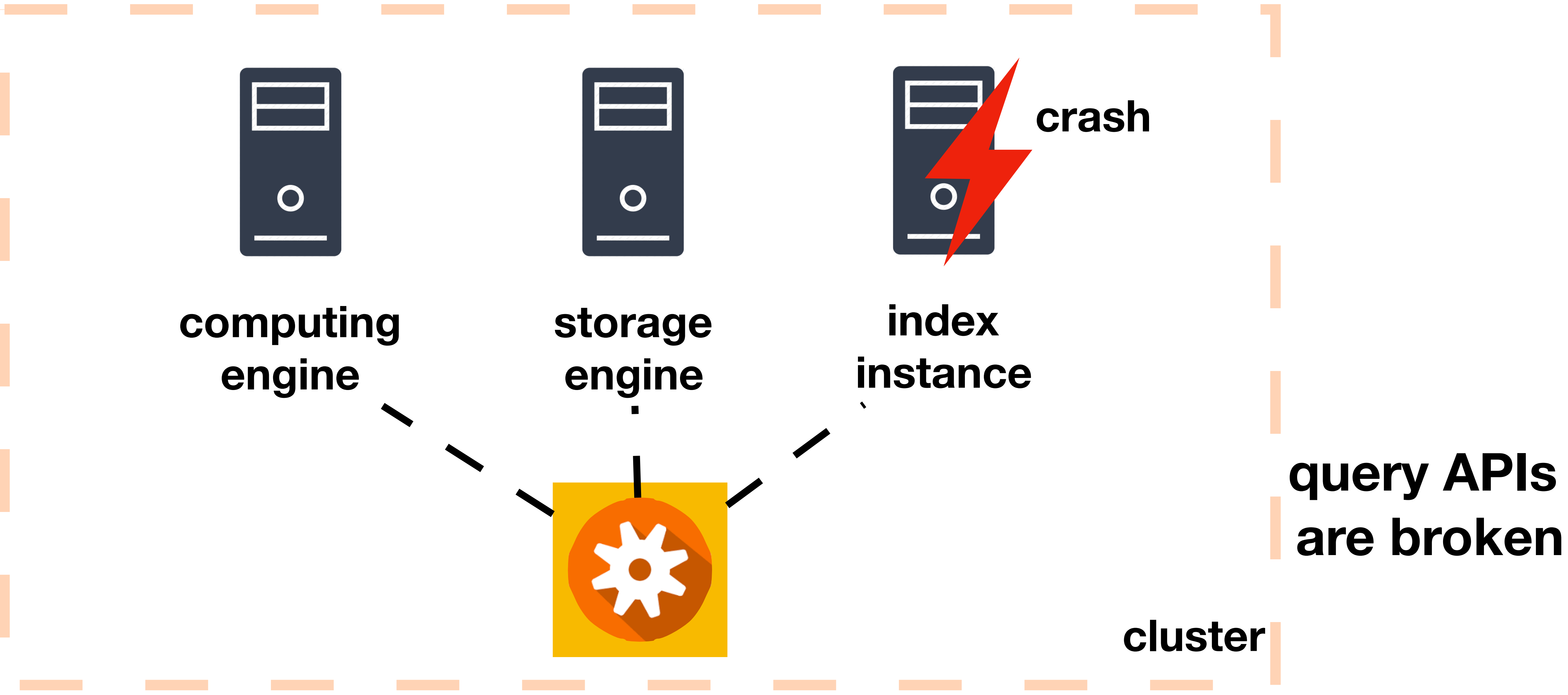
Software	Language	Cases	Versions	Date Range
ZooKeeper	Java	20	17 (3.2.1–3.5.3)	12/01/2009–08/28/2018
Cassandra	Java	20	19 (0.7.4–3.0.13)	04/22/2011–08/31/2017
HDFS	Java	20	14 (0.20.1–3.1.0)	10/29/2009–08/06/2018
Apache	C	20	16 (2.0.40–2.4.29)	08/02/2002–03/20/2018
Mesos	C++	20	11 (0.11.0–1.7.0)	04/08/2013–12/28/2018

Study scope

A partial failure is, in a process π to be when a fault does not crash π but causes safety or liveness violation or severe slowness for some functionality $R_f \subsetneq R$



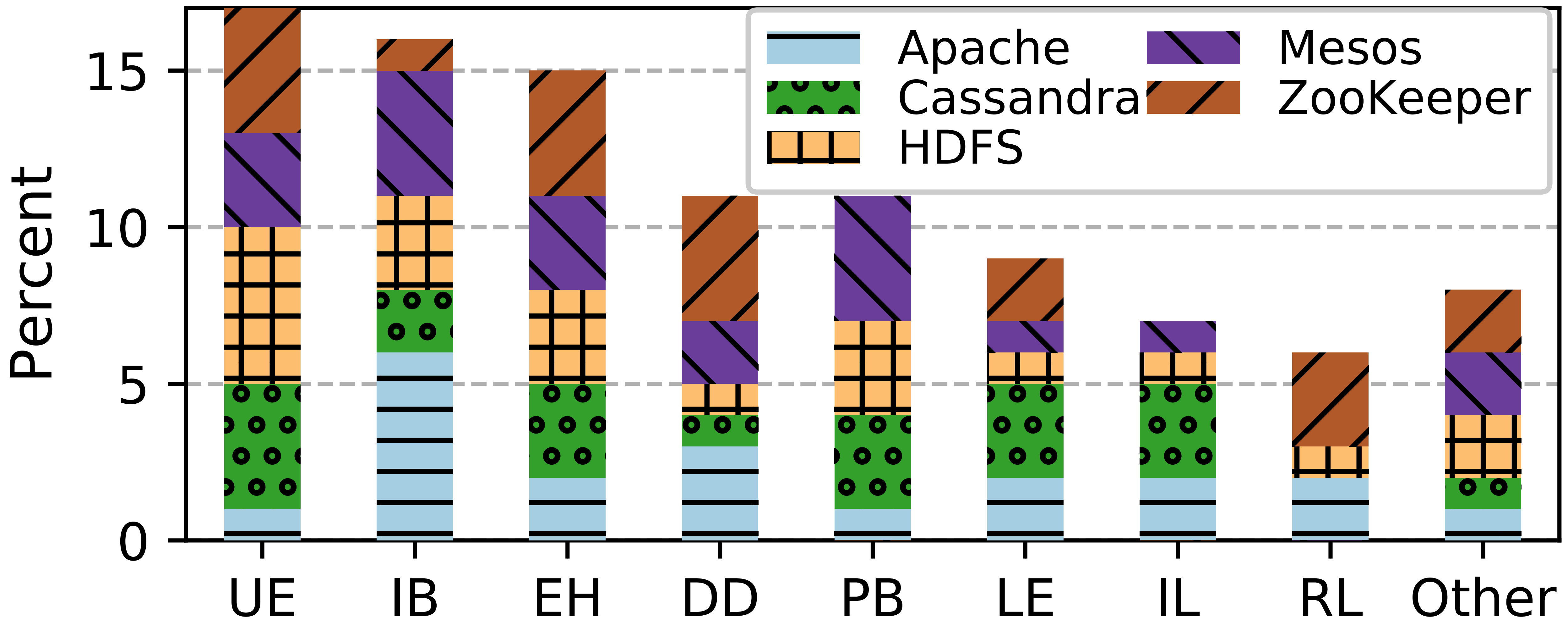
Process-level



Service-level

Finding 1: root causes are diverse

- **There is no single uniformed or dominating root cause.**
- **The top three (total 48%) root cause types are uncaught errors, indefinite blocking, and buggy error handling.**



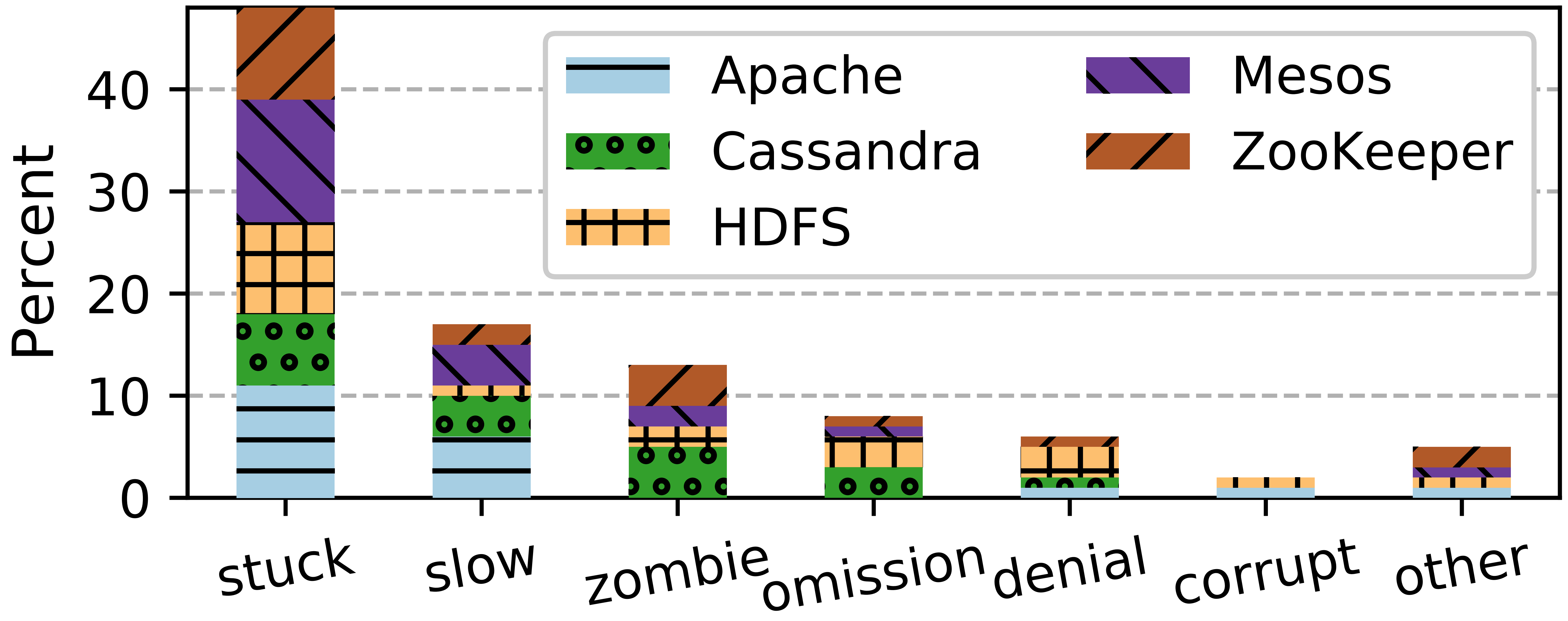
Root cause distribution.

UE: uncaught error; IB: indefinite blocking; EH: buggy error handling;
DD: deadlock; PB: performance bug; LE: logic error; IL: infinite loop; RL: resource leak.

Finding 2: nearly half cases cause stuck issues

- Nearly half (48%) of the partial failures cause some functionality to be stuck.

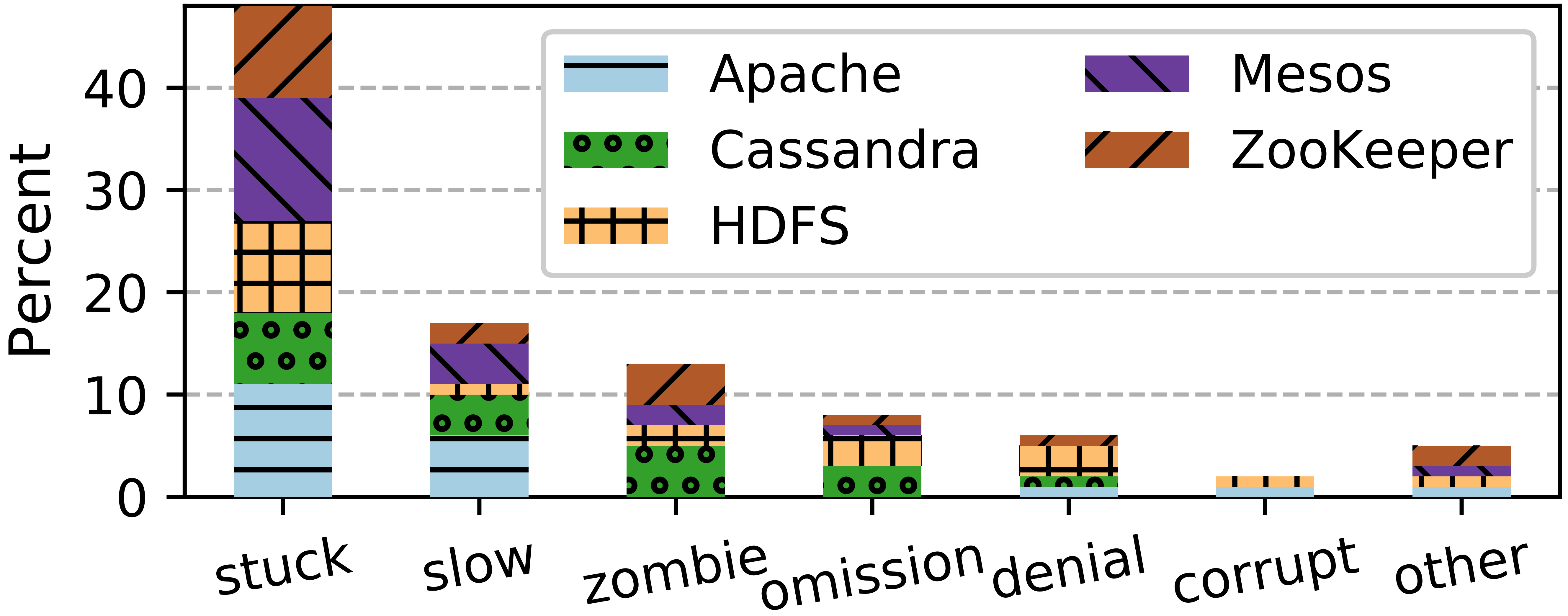
- 17% of the partial failures causes certain operations to take a long time to complete.



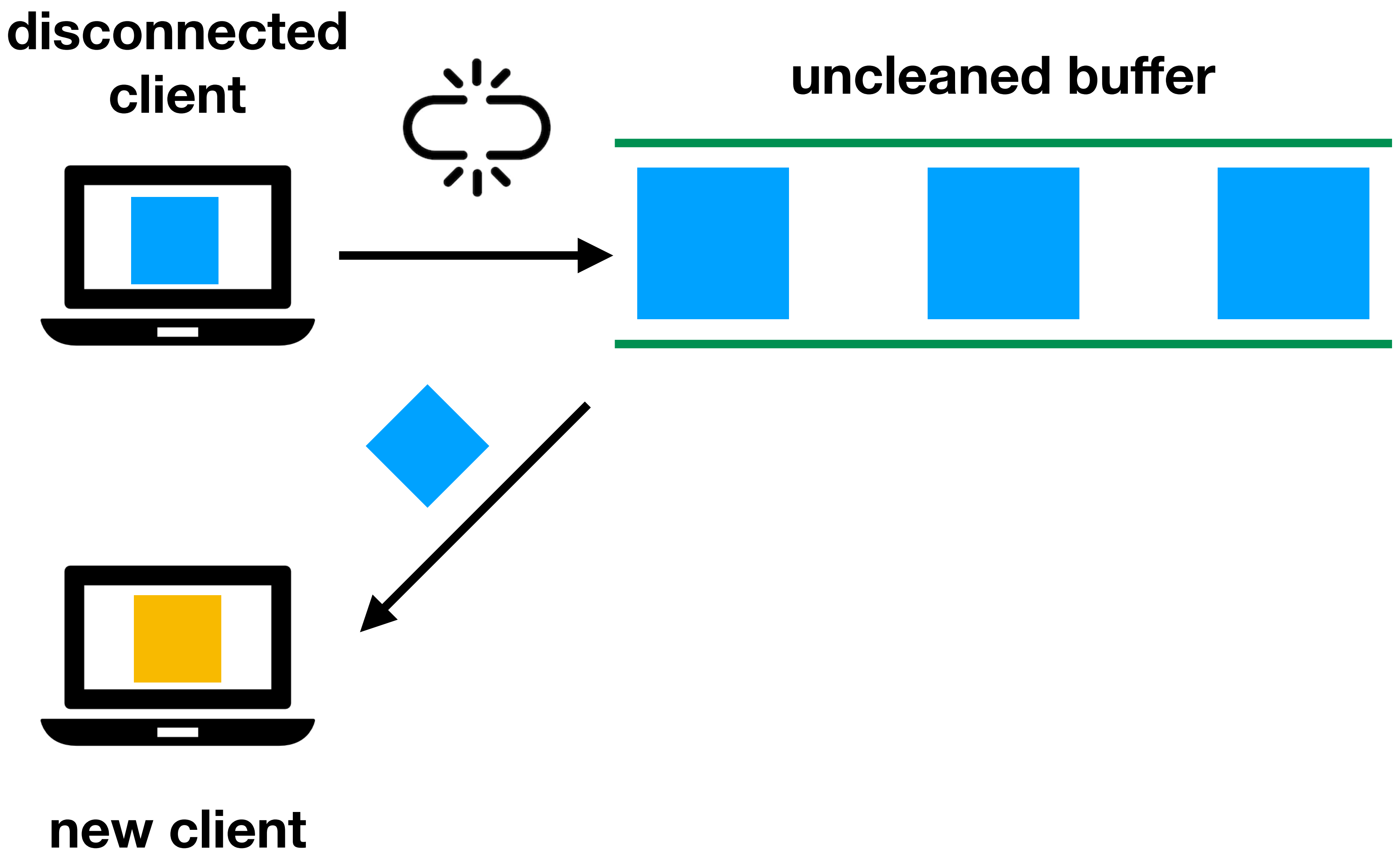
Consequence of studied failures.

Finding 3: 15% cases are totally silent

- 15% of the partial failures are silent (including data loss, corruption, inconsistency, and wrong results).



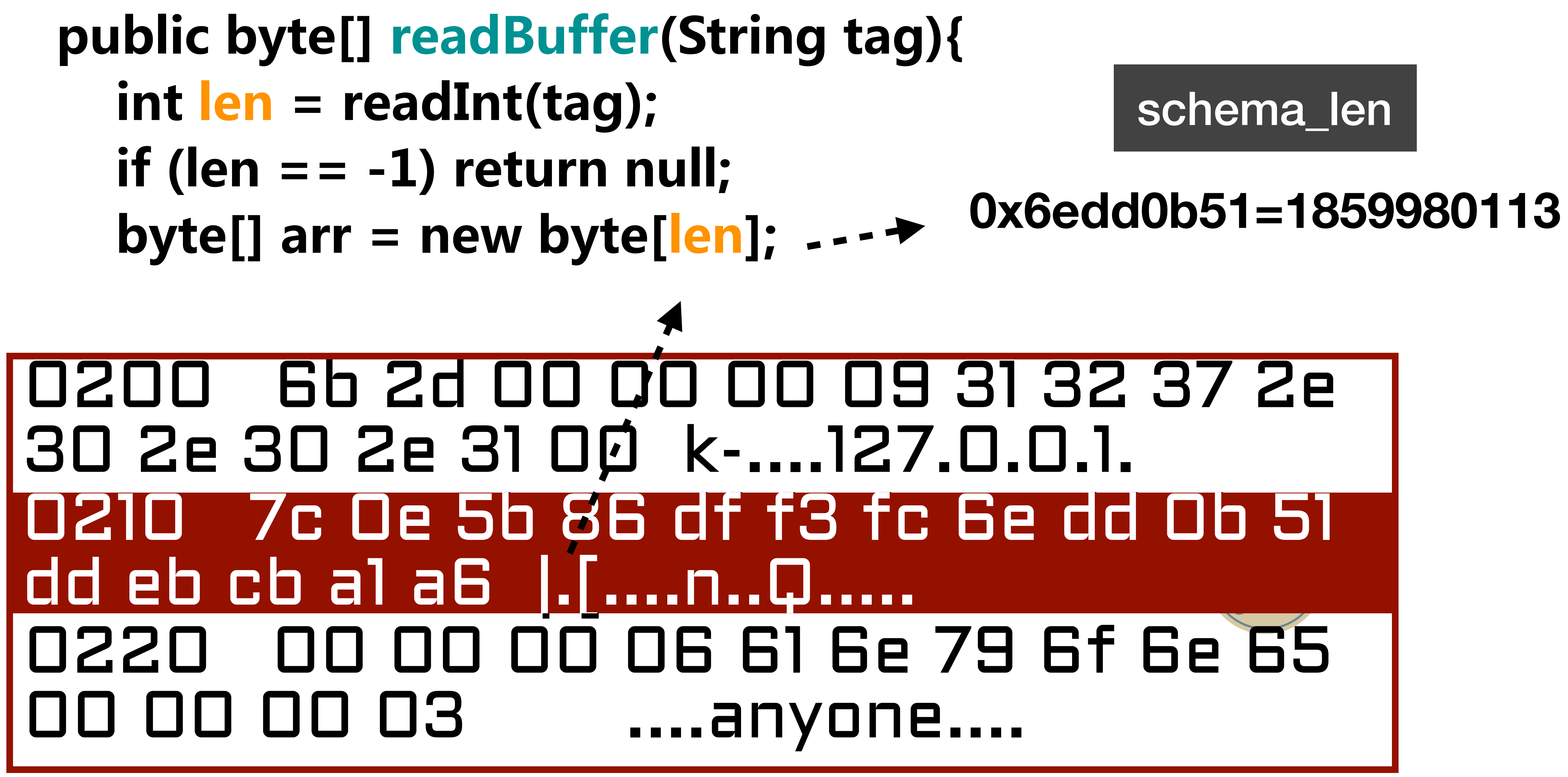
Consequence of studied failures.



APACHE-53727

Finding 4: most cases are triggered by unique production workload or environment

- 71% of the partial failures are triggered by some specific environment condition, or special input in the production.



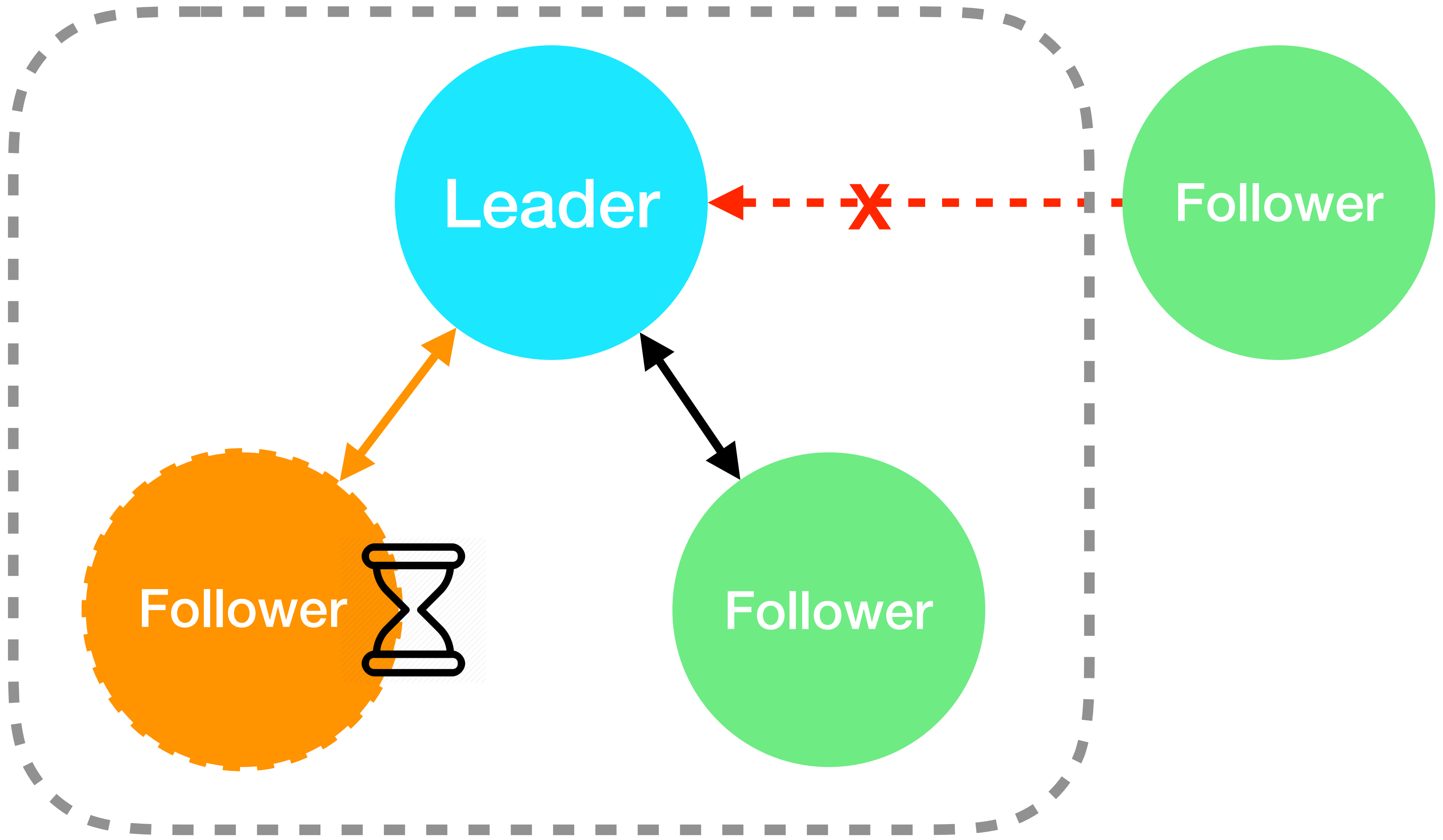
```
public byte[] readBuffer(String tag){
    int len = readInt(tag);
    if (len == -1) return null;
    byte[] arr = new byte[len];
```

schema_len

0x6edd0b51=1859980113

```
0200   6b 2d 00 00 00 09 31 32 37 2e
30 2e 30 2e 31 00  k-....127.0.0.1.
0210   7c 0e 5b 86 df f3 fc 6e dd 0b 51
dd eb cb a1 a6  |.[....n..Q.....
0220   00 00 00 06 61 6e 79 6f 6e 65
00 00 00 03       ....anyone....
```

ZK-602



ZK-914

Finding 5: debugging time is long

- **The median diagnosis time is 6 days and 5 hour.**
  - ✦ confusing symptoms of the failures mislead the diagnosis direction
  - ✦ insufficient exposure of runtime information in the faulty process

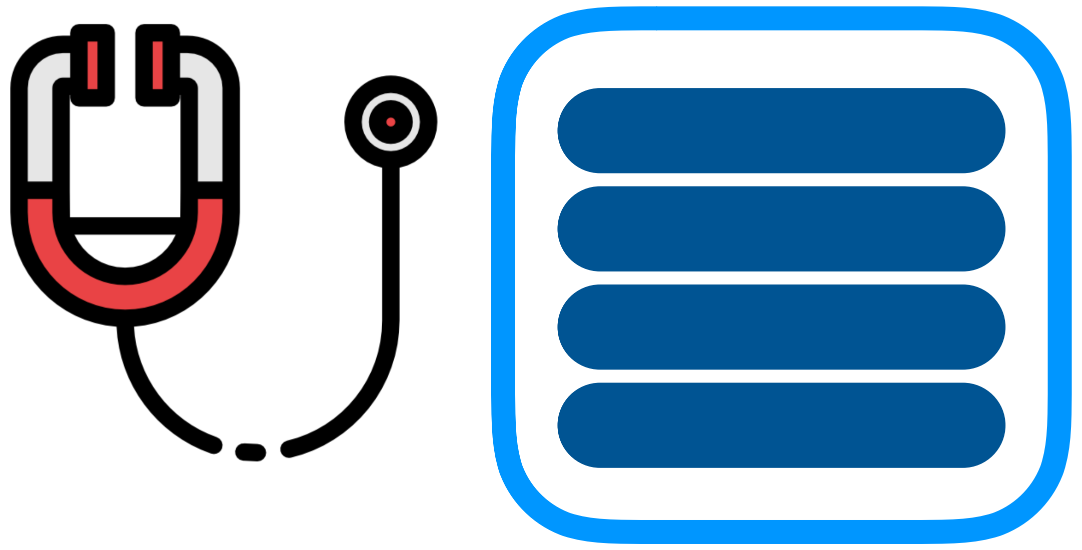
DEBUG

INFO

WARN

ERROR

enable DEBUG log



analyze heap

```
oa.writeInt(longKeyMap.size(),  “map");
+ System.out.println(val.getKey());
oa.writeLong(val.getKey(), "long");
```

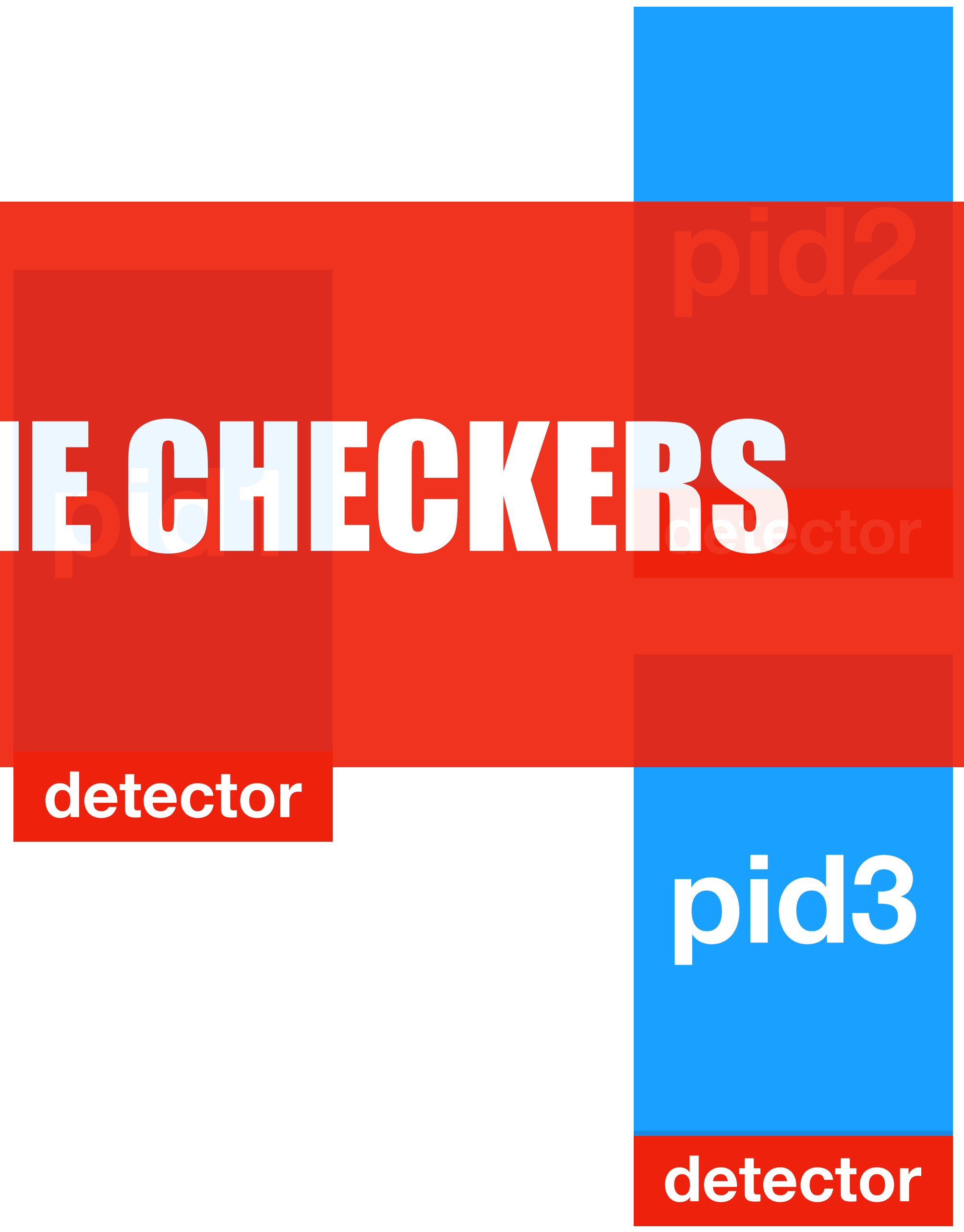
instrument code

How to deal with partial failures

Static Approach

TestSet1
TestSet2
TestSet3

Dynamic Approach



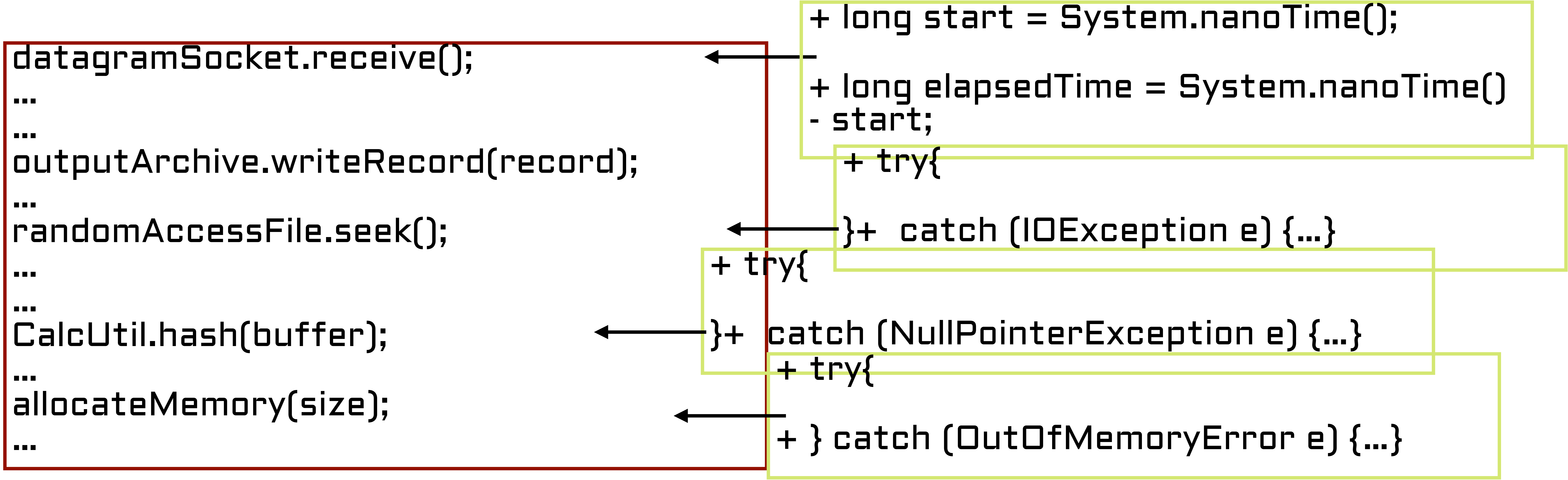
WE NEED MORE POWERFUL RUNTIME CHECKERS

no unique production env/workload

existing detectors are too shallow

Manual vs generated checkers

- Ask developers to manually add defensive checks?



```
datagramSocket.receive();
...
...
outputArchive.writeRecord(record);
...
randomAccessFile.seek();
...
...
CalcUtil.hash(buffer);
...
allocateMemory(size);
...
```

```
+ long start = System.nanoTime();

+ long elapsedTime = System.nanoTime()
- start;
```

```
+ try{

}+  catch (IOException e) {…}
```

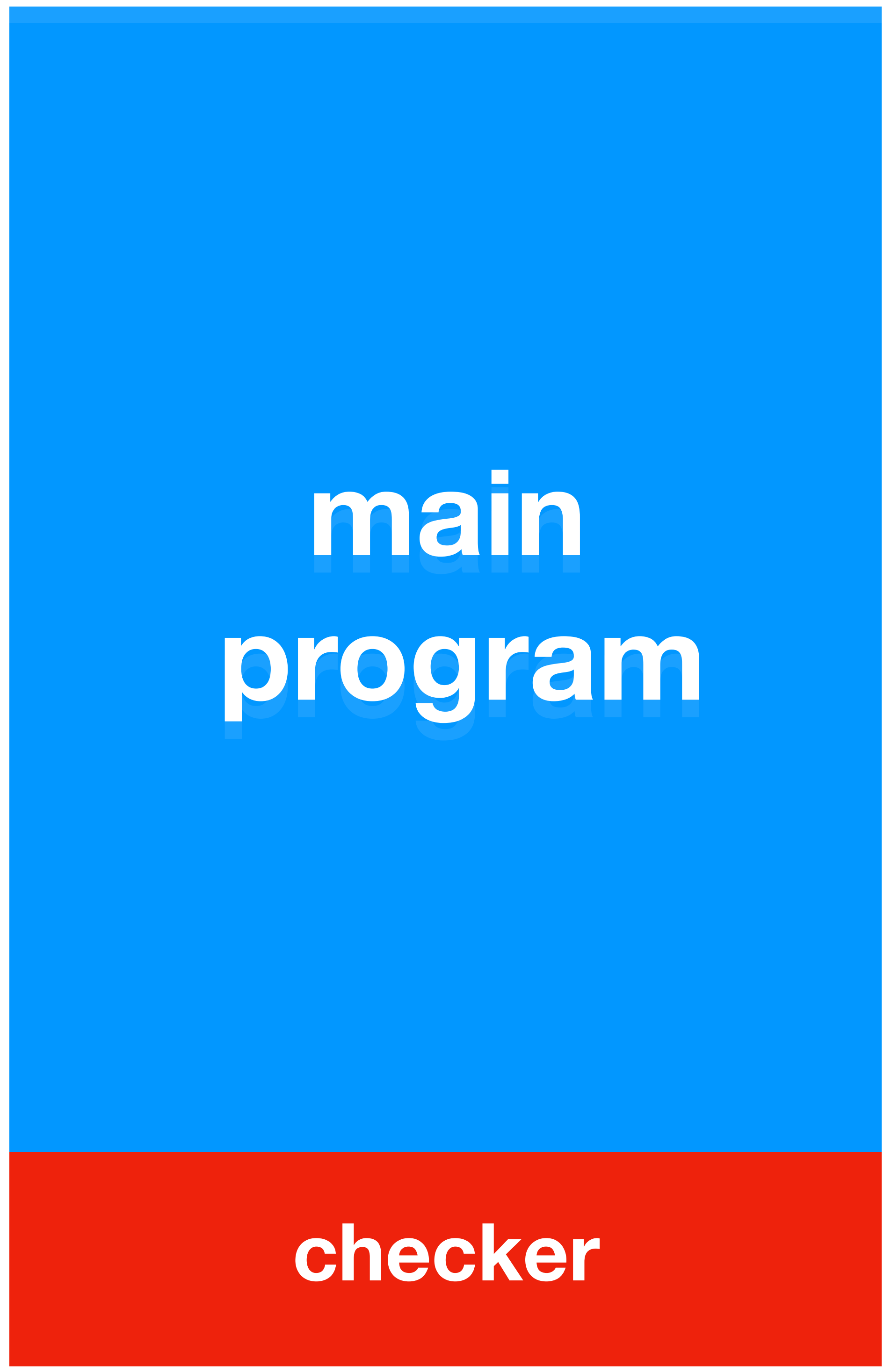
```
+ try{

}+  catch (NullPointerException e) {…}
```

```
+ try{

+ } catch (OutOfMemoryError e) {…}
```

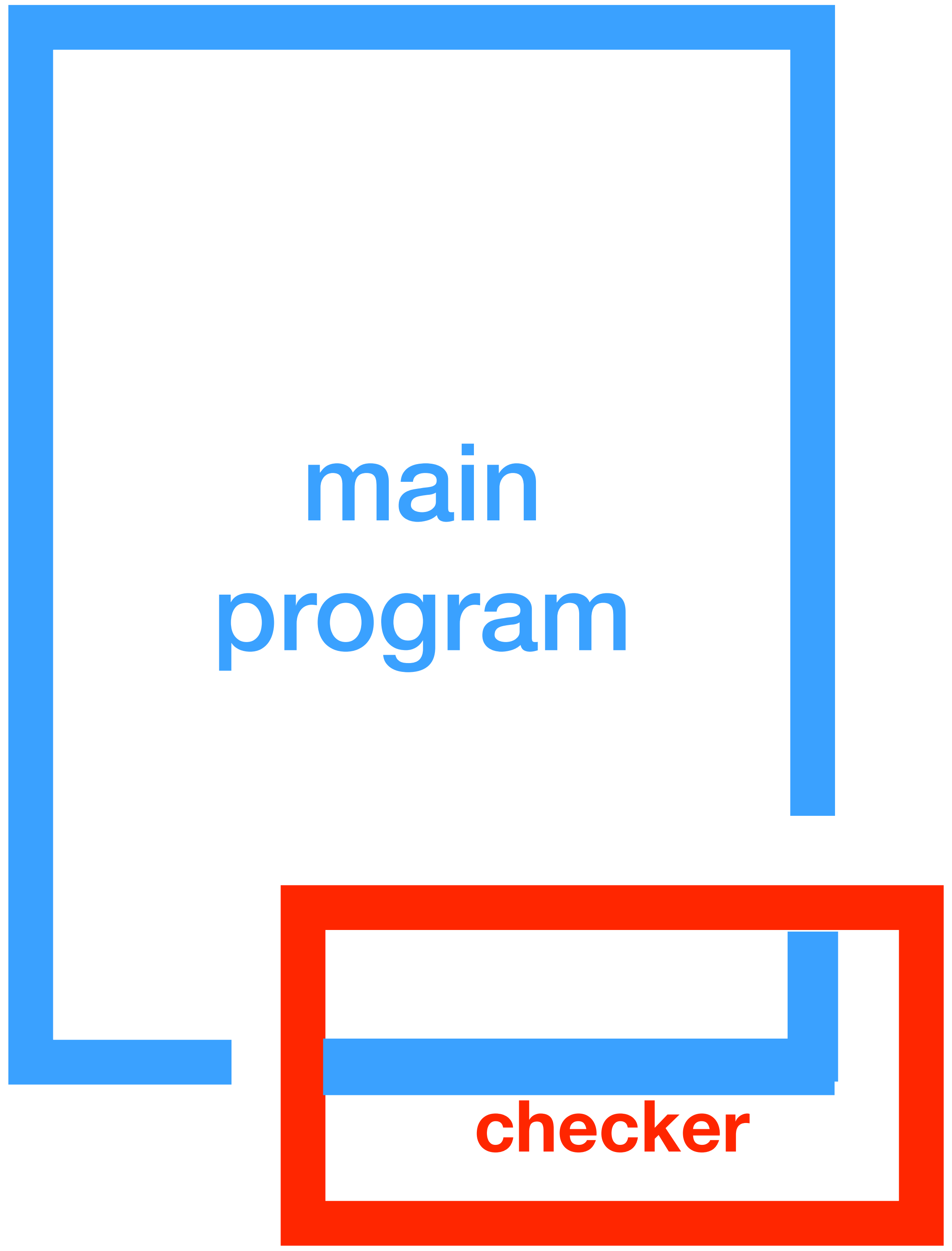
Manual vs generated checkers

- **Systematically** generate checkers to ease developers' burden
  - challenge: difficult to automate for all cases
  - opportunity: most of partial failures do not rely on deep semantic understanding to detect, such checkers can potentially be automatically constructed

Design principle: checkers intersect with the execution of a monitored process

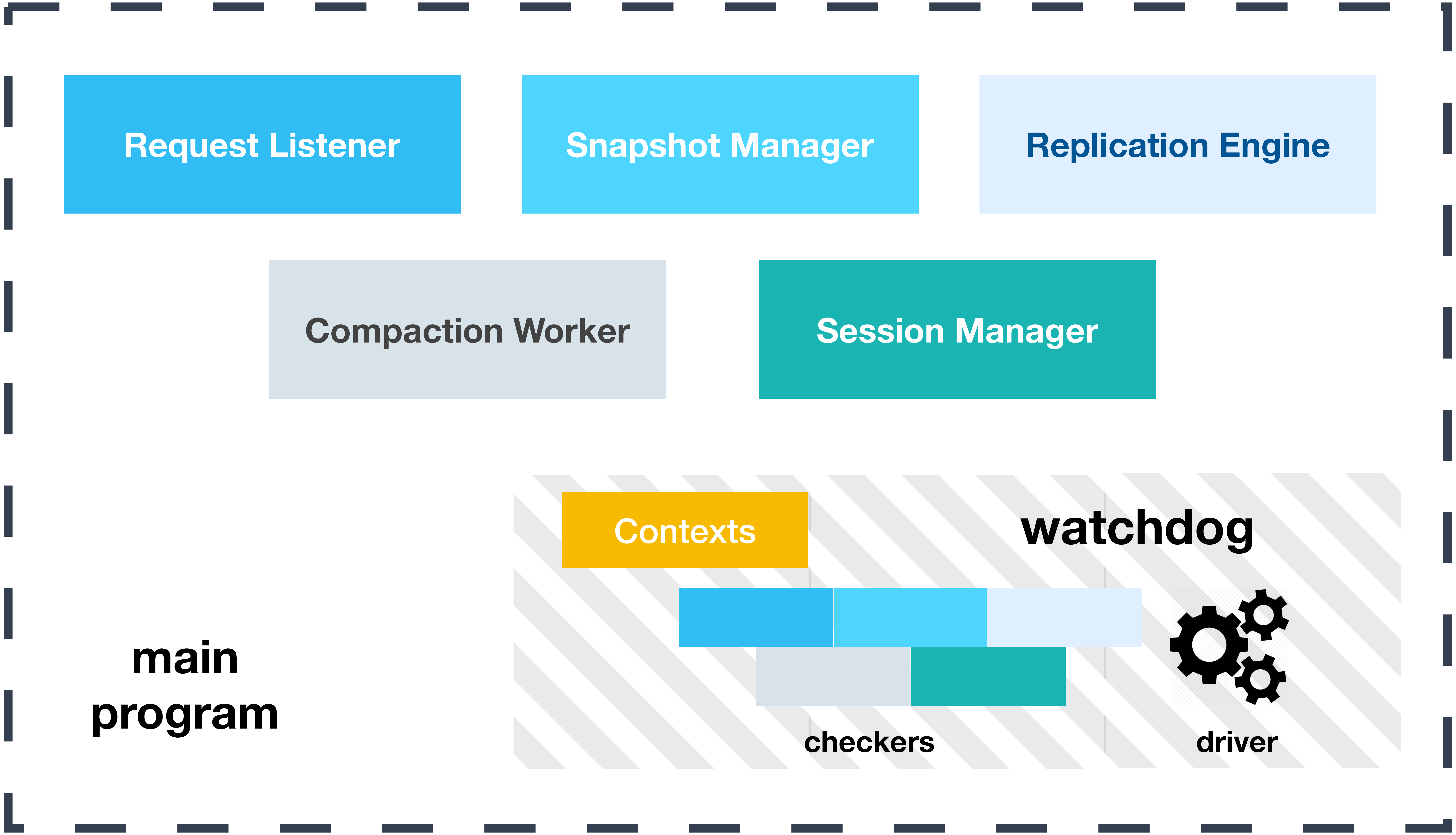


existing approach

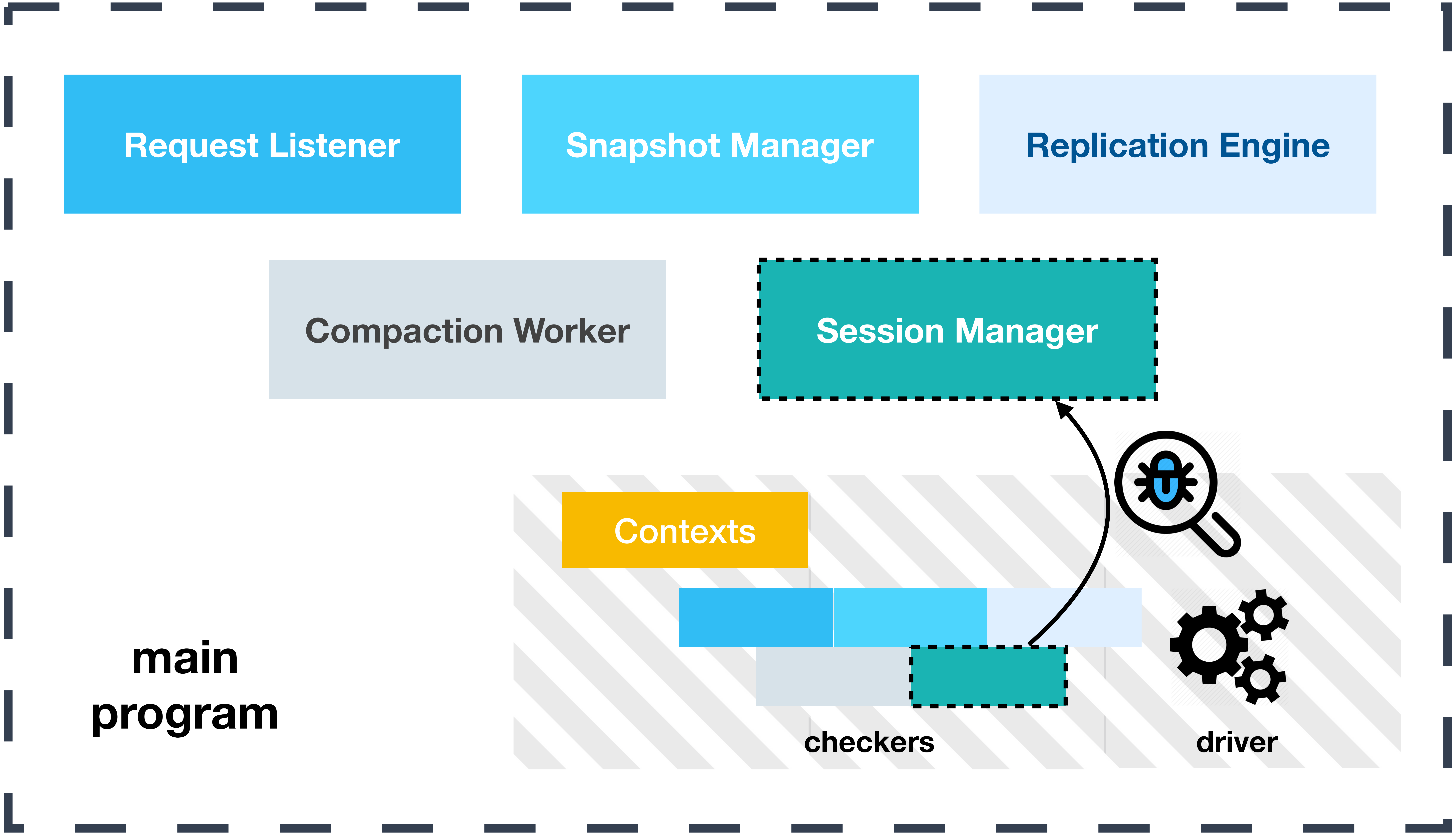


our approach

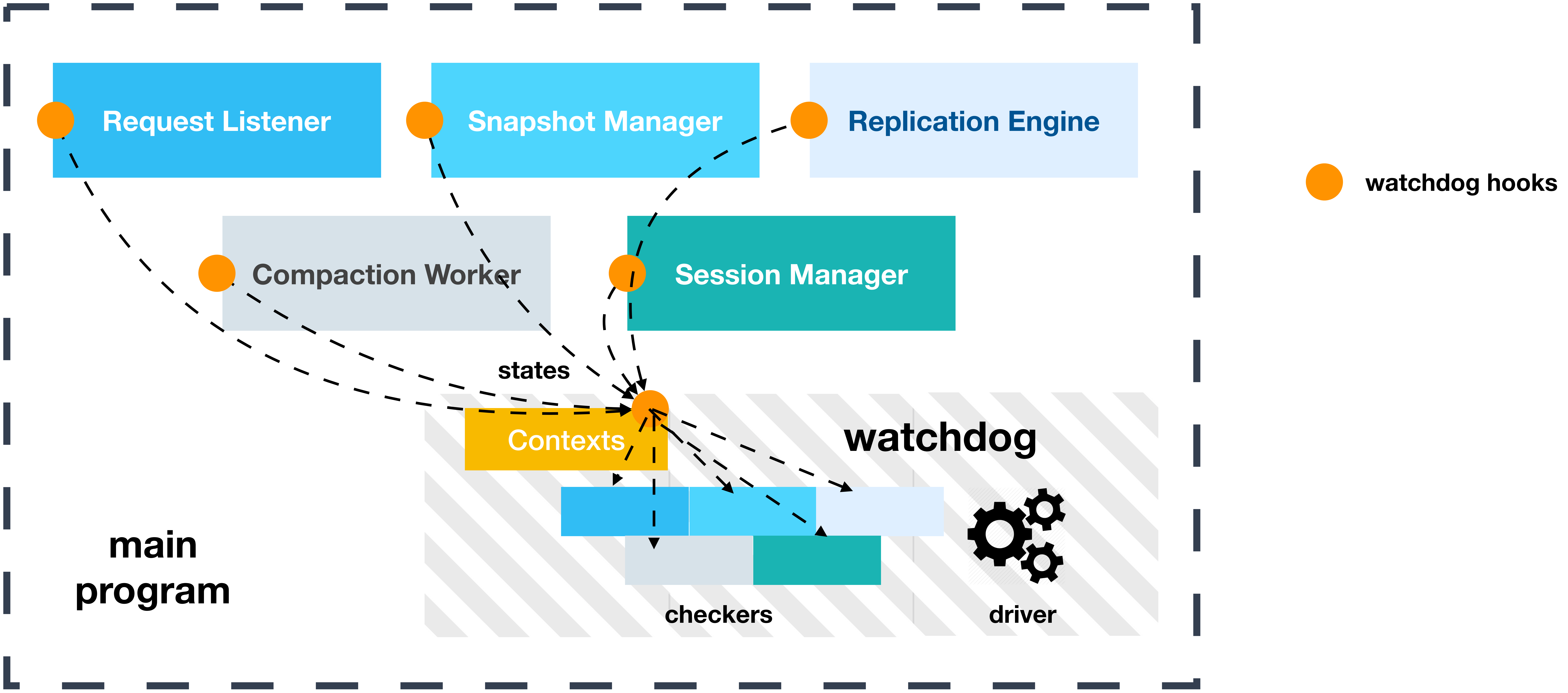
Intrinsic watchdog: Runtime



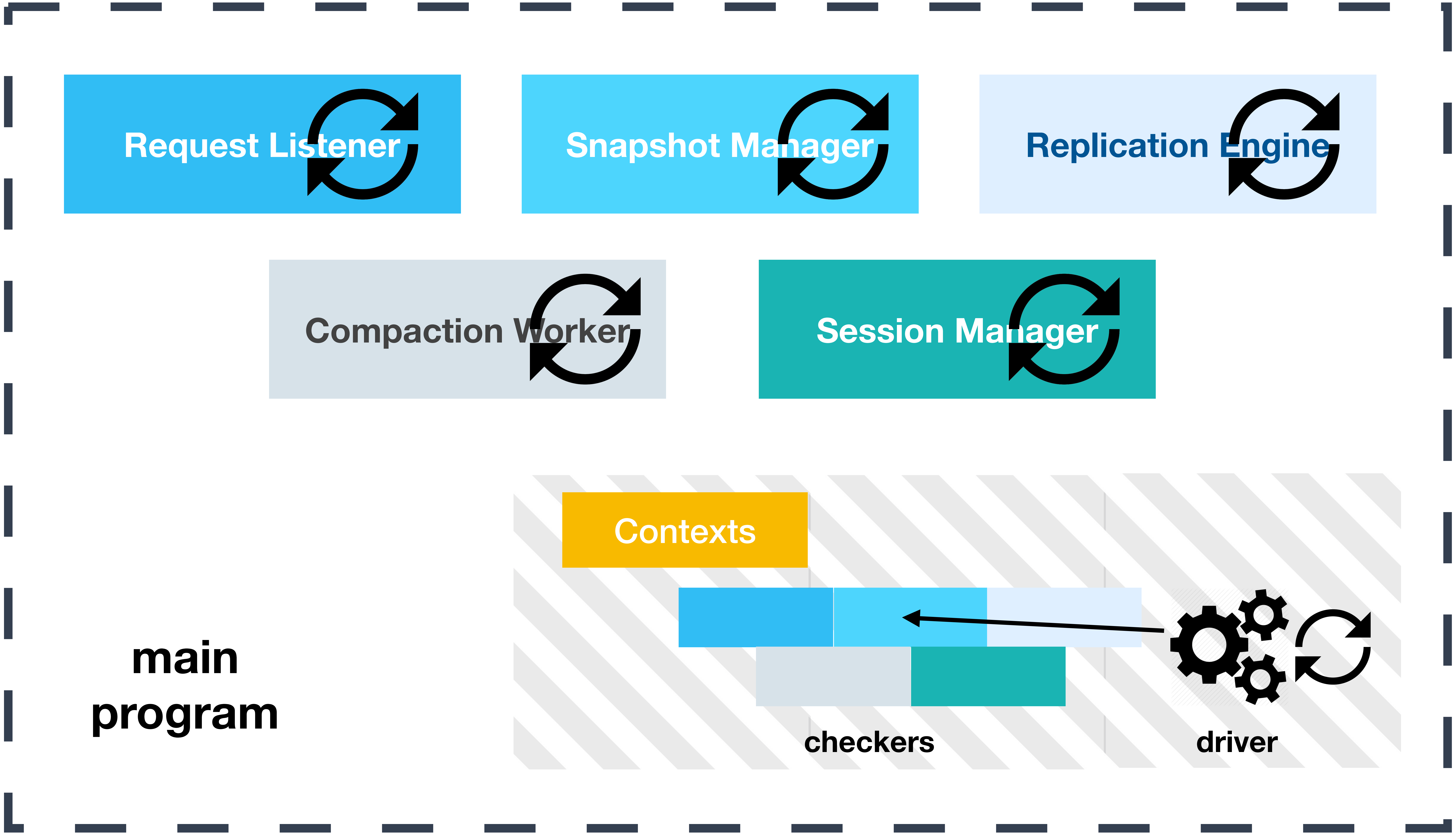
Characteristic I: customized



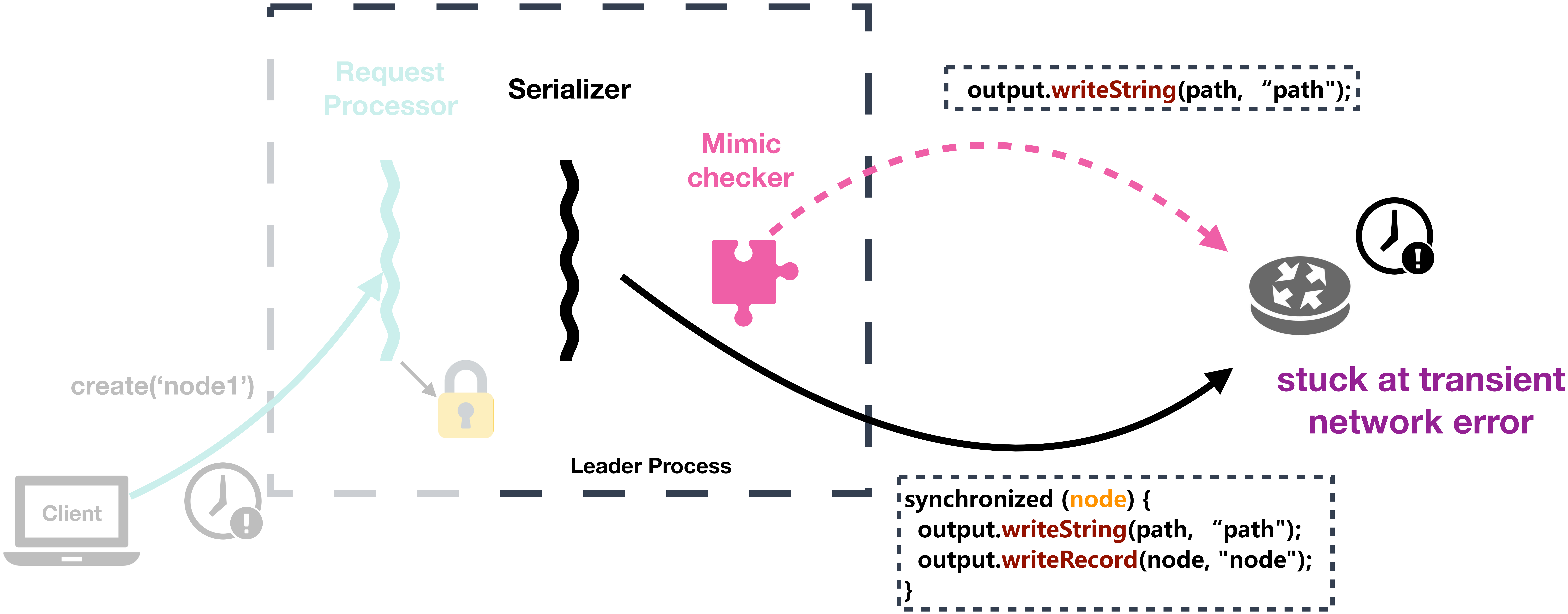
Characteristic II: stateful



Characteristic III: concurrent



Core idea: mimic checking

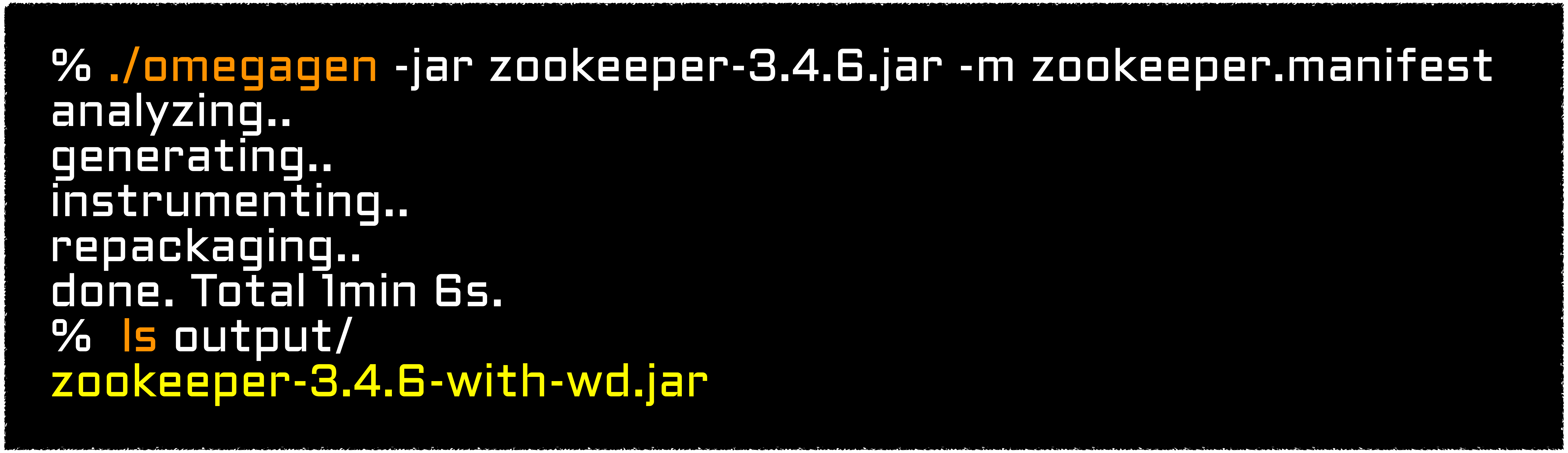


```
output.writeString(path, “path");
```

```
synchronized (node) {
  output.writeString(path, “path");
  output.writeRecord(node, "node");
}
```

Tool overview

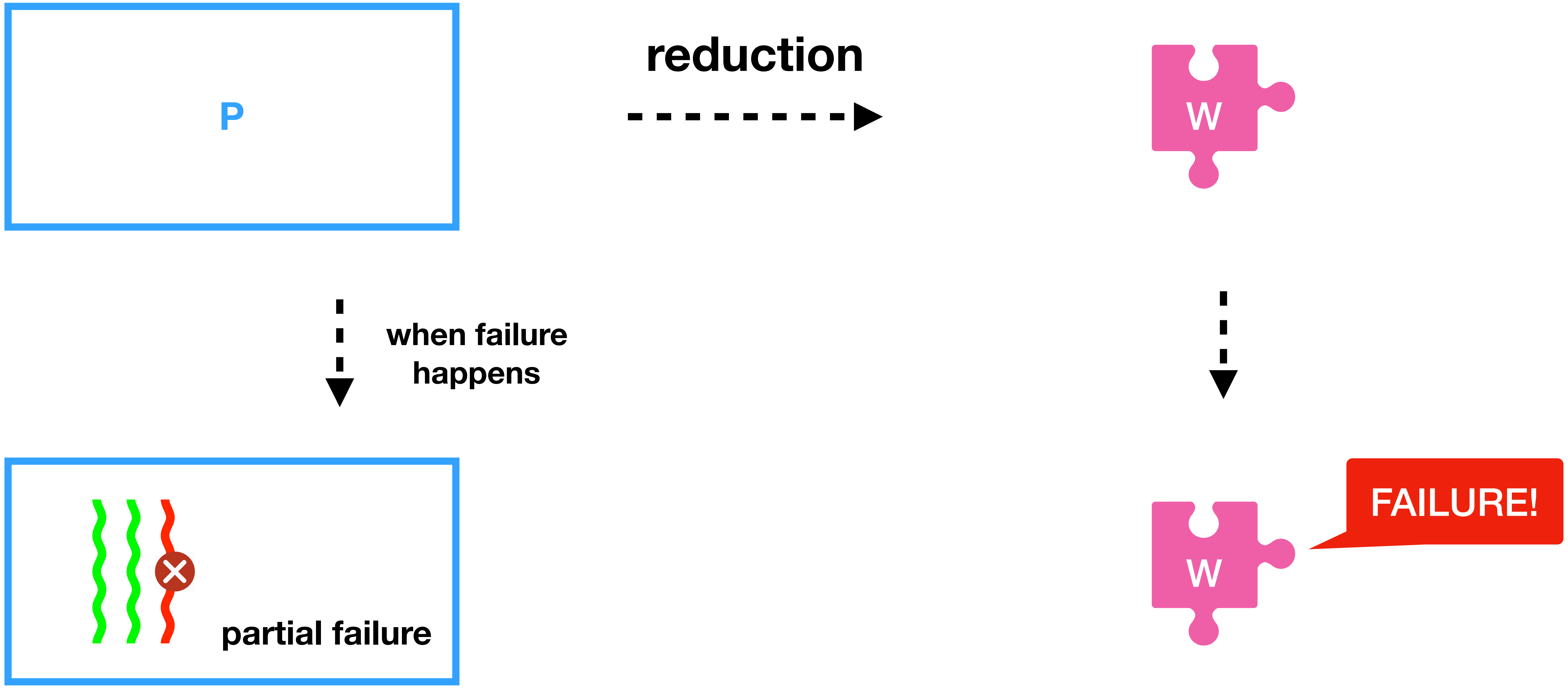
- OmegaGen
  - a prototype that systematically generate mimic-type watchdogs for system softwares
  - core technique: program reduction



```
% ./omegagen -jar zookeeper-3.4.6.jar -m zookeeper.manifest
analyzing..
generating..
instrumenting..
repackaging..
done. Total 1min 6s.
%  ls output/
zookeeper-3.4.6-with-wd.jar
```

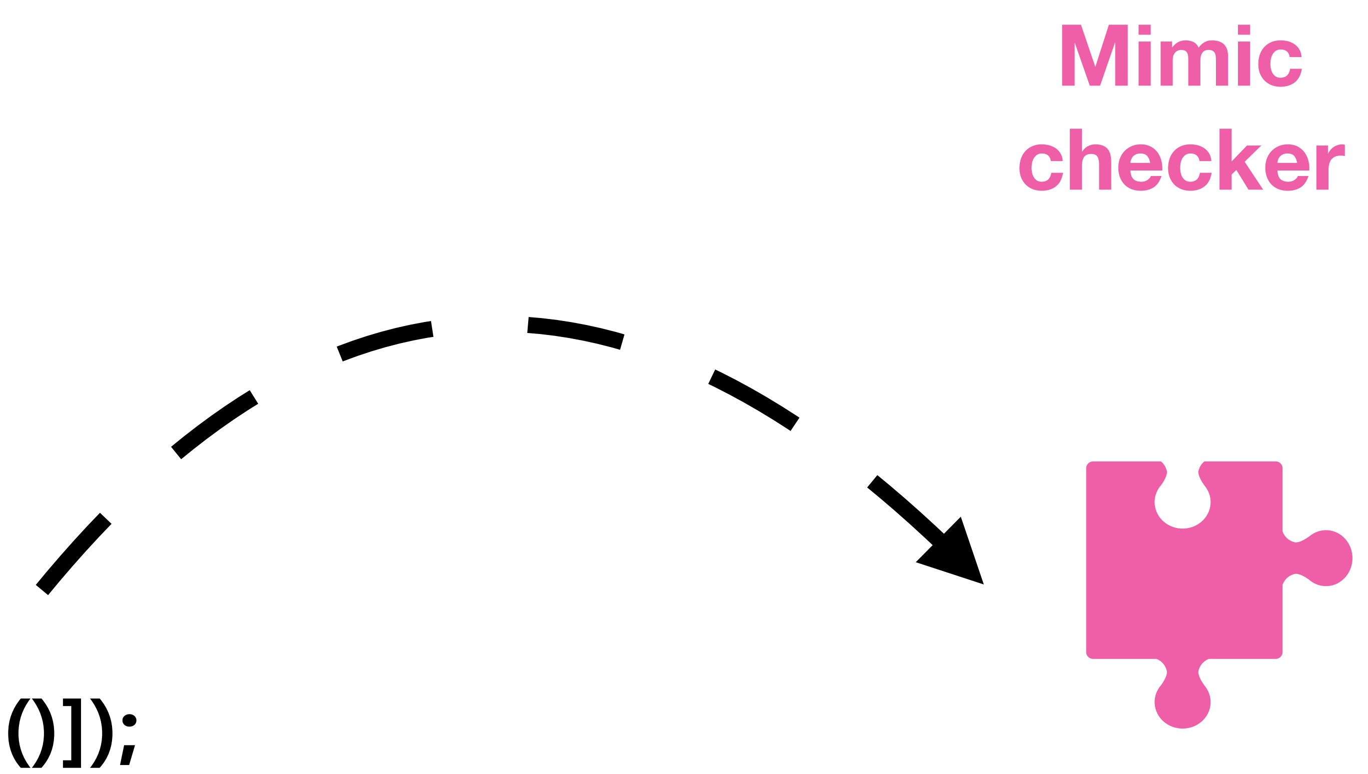
What is program reduction?

- Given a program P, create a watchdog W that can detect partial failures in P without imposing on P's execution.



1) We should not put everything into checker

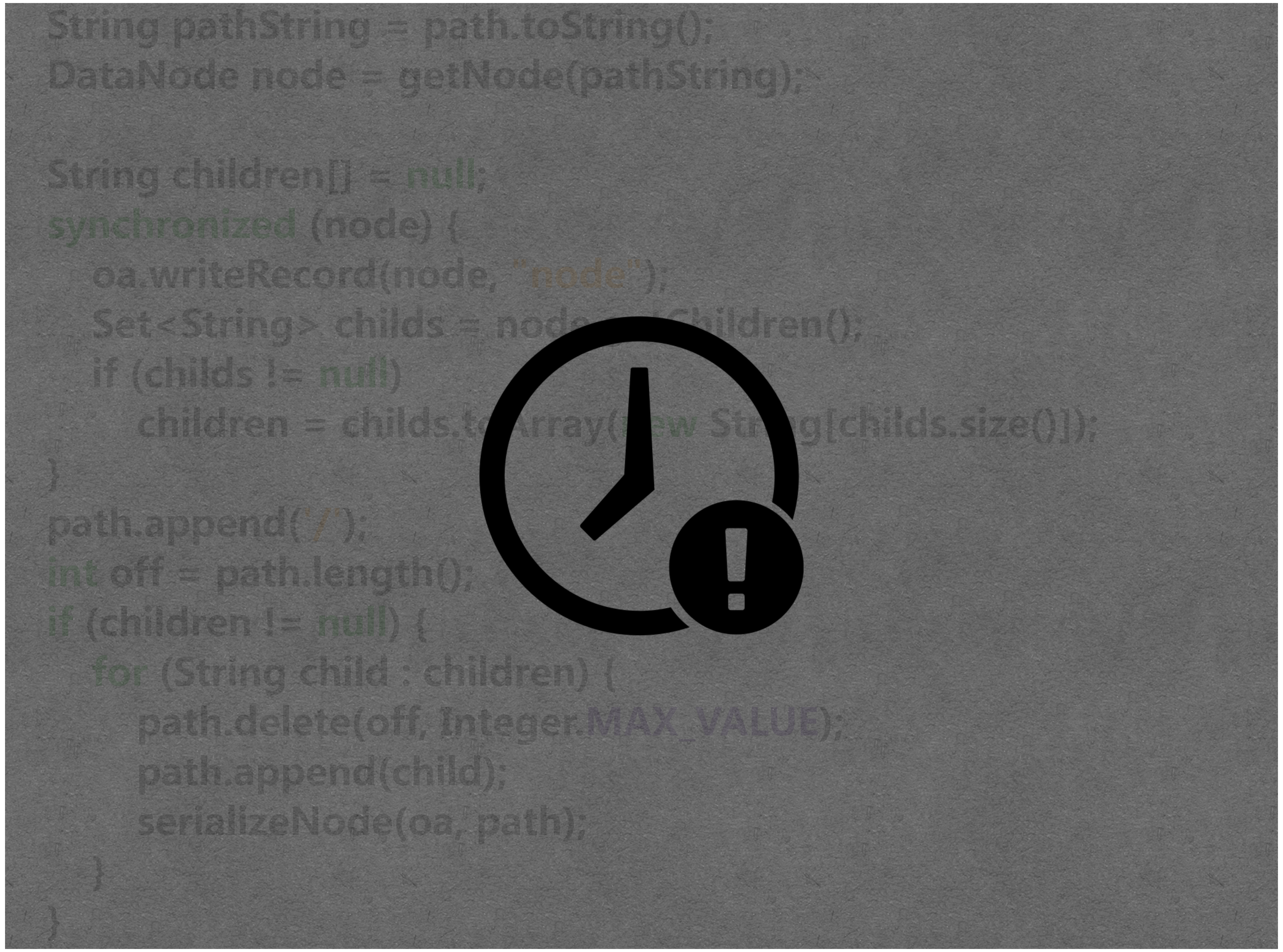
```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```



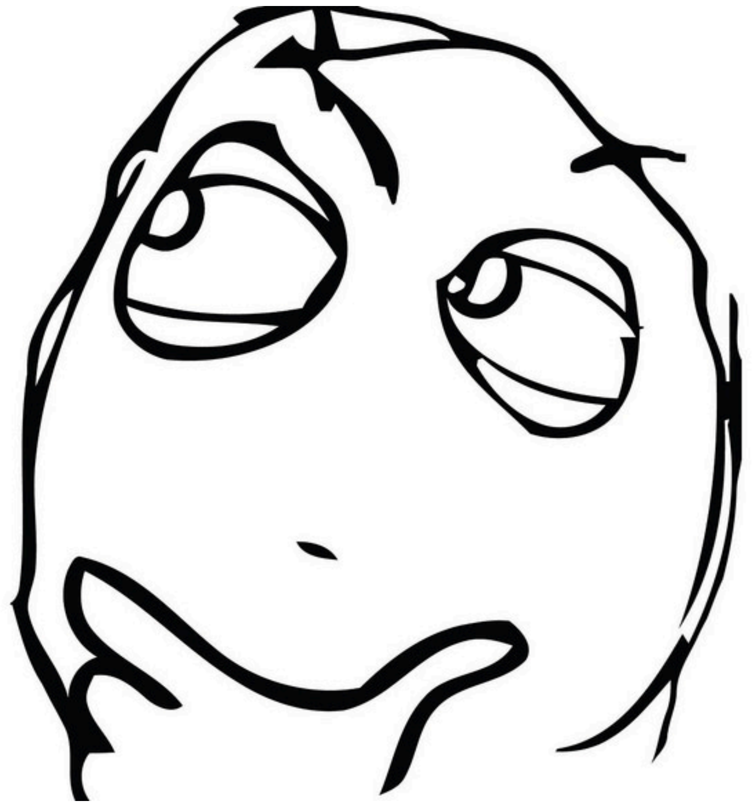
Mimic checker

what if put the whole snapshot operation into the checker and run?

1) We should not put everything into checker



```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```



checker can detect the timeout,
but we don’t know which part goes wrong

2) We need not put everything into checker

```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```

2) We need not put everything into checker

```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```

convert string

convert array

append path

iterate children
and modify path

**a lot of operations are logically deterministic
and should be checked before production**

2) We need not put everything into checker

```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    if (children != null) {
        for (String child : children) {
            path.delete(off, Integer.MAX_VALUE);
            path.append(child);
            serializeNode(oa, path);
        }
    }
}
```

do I/O + in synchronized block

some operations are more vulnerable in the production environment

Program reduction

- **Five steps**
  - #1 locate long-running regions
  - #2 reduce the program
  - #3 locate vulnerable operations
  - #4 encapsulate watchdog checkers
  - #5 insert watchdog hooks

Step#1 locate long-running regions

initialization stage

long-running stage

cleanup stage

```
public class SyncRequestProcessor {
 public void run() {
  int logCount = 0;

  setRandRoll(r.nextInt(snapCount/2));
  ...
  while (running) {
   ...
   if (logCount > (snapCount / 2 ))
    zks.takeSnapshot();
  }
  ..
  LOG.info("SyncRequestProcessor exited!");
}
```

entry list

org.apache.zookeeper.server.ZooKeeperServer.takeSnapshot()
...

Step#2 reduce the program

```
public class SyncRequestProcessor {
 public static void serializeSnapshot(DataTree dt, ...) {

   ...
   dt.serialize(oa, “tree”);                        keep reducing
 }
}
public class DataTree{
 public void serialize(OutputArchive oa, String tag) {
   scout = 0;
   serializeNode(oa, new StringBuilder(“”));        keep reducing
   ...
}
```

Step#3 locate vulnerable operations

```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);

    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    ...
}
```

vulnerable op found, mark

our heuristic

I/O,
synchronization, resource,
communication related
method invocations,
...

Step#4 encapsulate watchdog checkers

```
public class SyncRequestProcessor$Checker {
 public static void serializeNode_reduced(OutputArchive arg0, DataNode arg1) {
  try{
   arg0.writeRecord(arg1, “node”);
  } catch (Throwable ex)
  …
 }
 public static Status checkTargetFunction0() {
  …
  Context ctx = ContextFactory.serializeNode_reduced_context();
  if (ctx.status == READY) {
   OutputArchive arg0 = ctx.args_getter(0);
   DataNode arg1 = ctx.args_getter(1);
   executor.runAsyncWithTimeout(serializeSnapshot_reduced(arg0, arg1), TIMEOUT);
  }
  else
   LOG.debug(“checker context not ready”);
  …
 }
}
```

extracted vulnerable operations

Step#5 insert watchdog hooks

```
void serializeNode(OutputArchive oa, StringBuilder path) throws IOException {
    String pathString = path.toString();
    DataNode node = getNode(pathString);
                                        + ContextFactory.serializeNode_context_setter(oa, node);
    String children[] = null;
    synchronized (node) {
        oa.writeRecord(node, "node");
        Set<String> childs = node.getChildren();
        if (childs != null)
            children = childs.toArray(new String[childs.size()]);
    }
    path.append('/');
    int off = path.length();
    ...
}
```

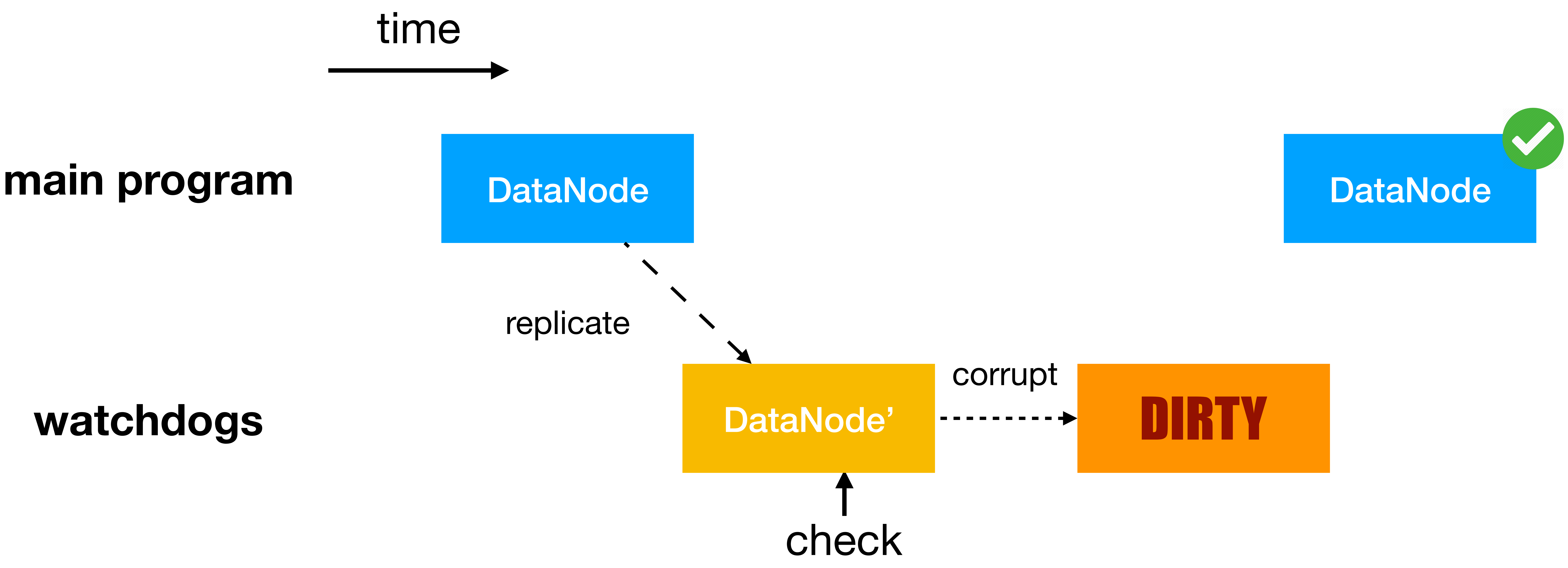
insert context hook before vulnerable operation

Validation

- **An error reported by a watchdog checker could be transient or tolerable.**
  - e.g. a transient network delay that caused no damage
- **Watchdog driver by default simply re-executes the checker and compare for transient errors.**
  - OmegaGen also allows developers write their own user-defined validation tasks to check some entry functions, e.g., processRequest(req)
  - The tool would automate the part of deciding which validation task to invoke depending on which checker failed.

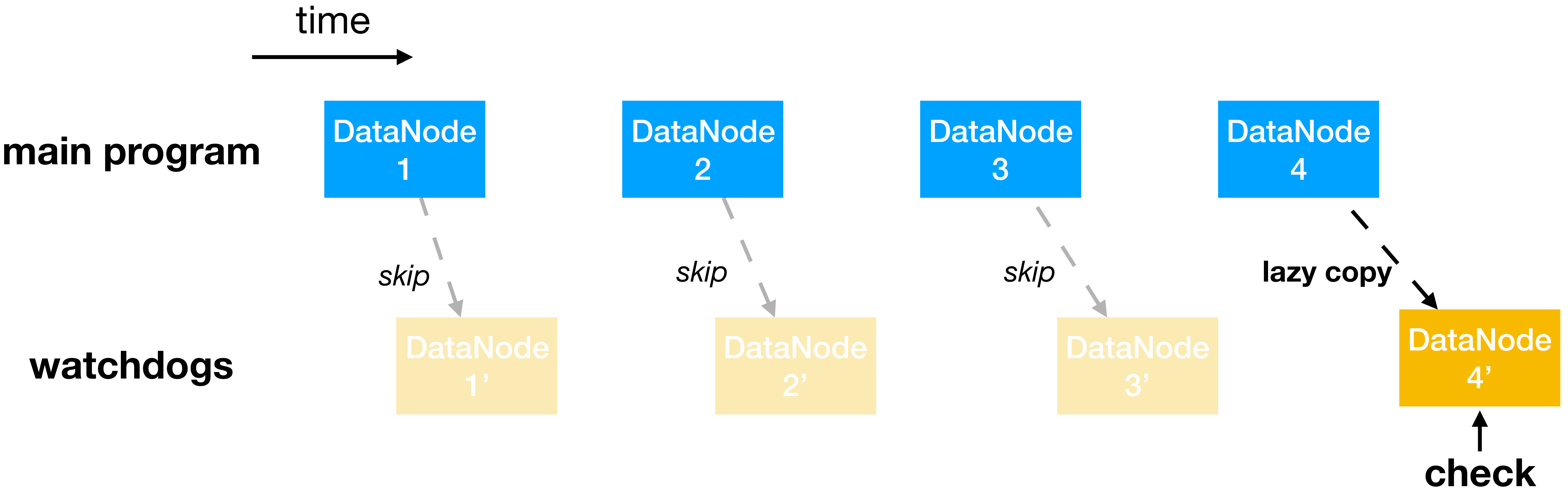
Prevent Side Effects

- **Context Replication (memory isolation)**
  - context manager will replicate the checker context so that any modifications are contained in the watchdog’s state



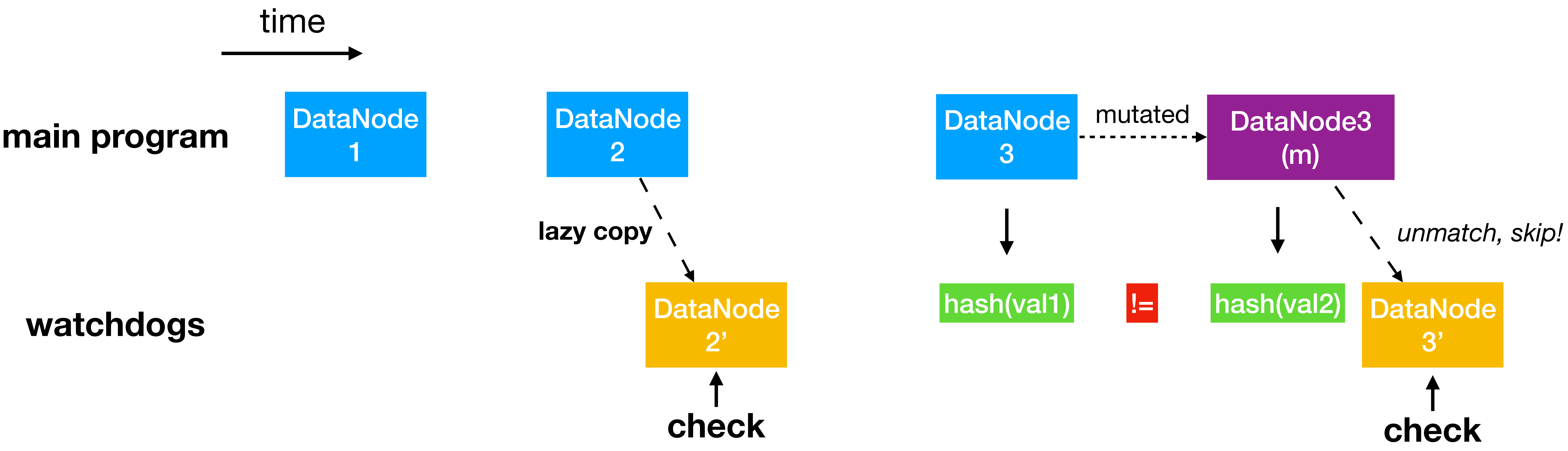
Prevent Side Effects

- **Context Replication (memory isolation)**
  - to reduce performance overhead: immutability analysis + lazy copy



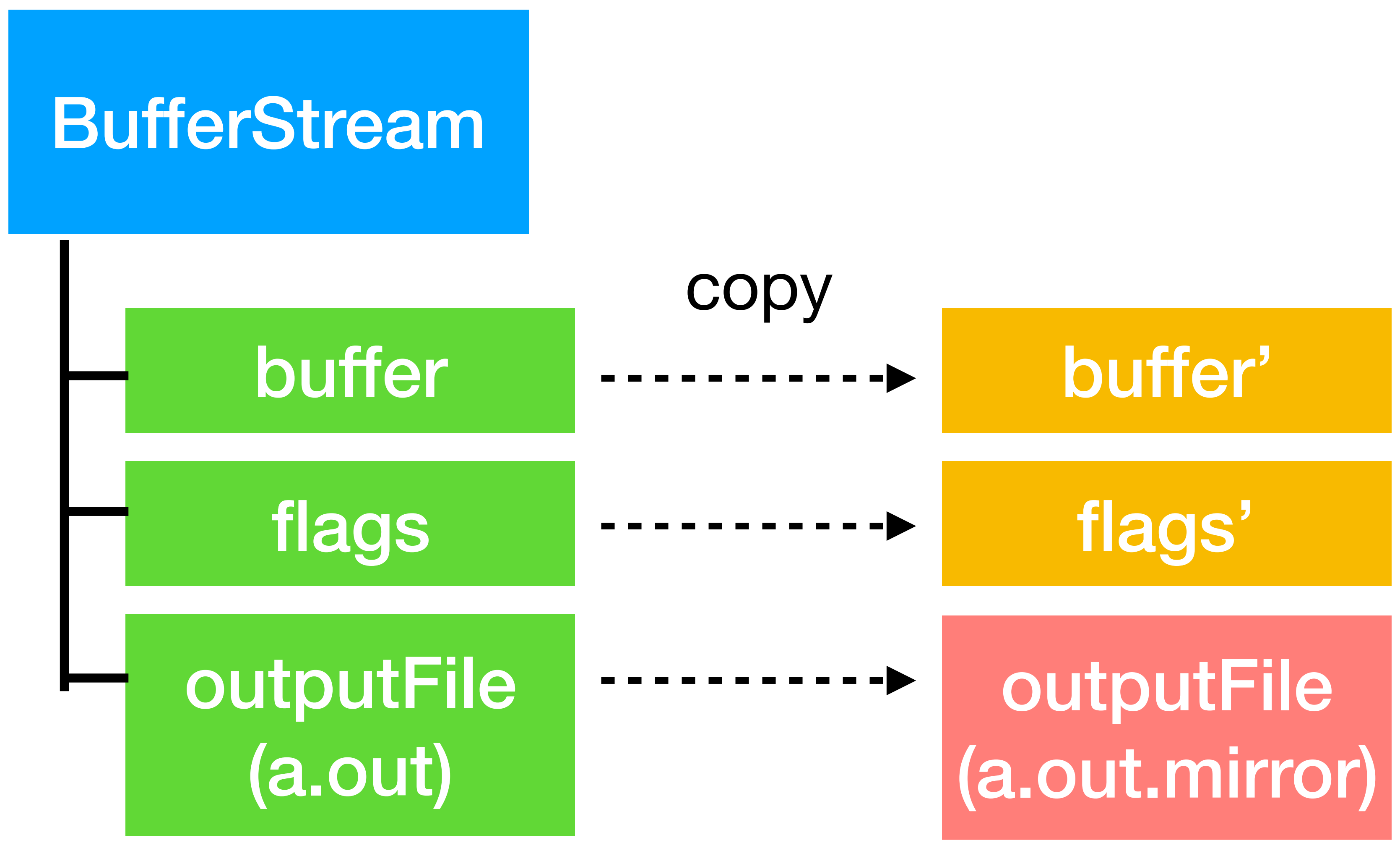
Prevent Side Effects

- **Context Replication (memory isolation)**
  - check consistency before copying and invocation with hashCode and versioning

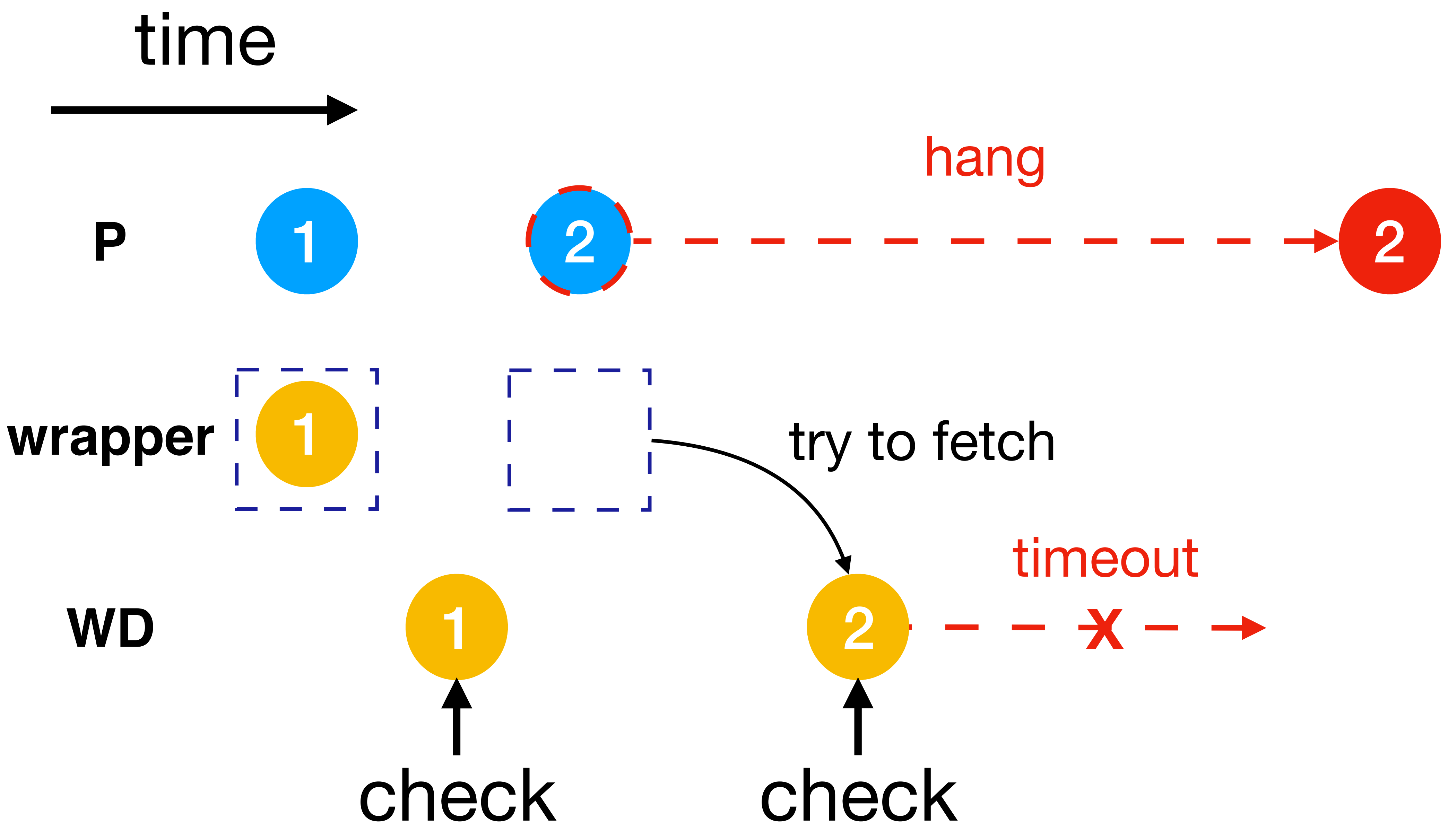


Prevent Side Effects

- I/O Redirection and Idempotent Wrappers (I/O isolation)
  - write: file-related resource replicated with target path changed to test file
  - read: watchdogs pre-read contexts and cache



write-redirection



read-redirection

Evaluation

Evaluated systems

	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
SLOC	28K	102K	219K	728K	191K	229K
Methods	3,562	12,919	79,584	179,821	16,633	10,432

Scale of evaluated system software

Watchdog generation

	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
Watchdogs	96	190	174	358	161	88
Methods	118	464	482	795	371	222
Operations	488	2,112	3,416	9,557	6,116	752

Number of watchdogs and checkers generated

22 real-world partial failures reproduced for evaluation

JIRA Id.	Id.	Root Cause	Conseq.	Sticky?	Study?
ZooKeeper-2201	ZK1	Bad Synch.	Stuck	No	Yes
ZooKeeper-602	ZK2	Uncaught Error	Zombie	Yes	Yes
ZooKeeper-2325	ZK3	Logic Error	Inconsist	Yes	No
ZooKeeper-3131	ZK4	Resource Leak	Slow	Yes	Yes
Cassandra-6364	CS1	Uncaught Error	Zombie	Yes	Yes
Cassandra-6415	CS2	Indefinite Blocking	Stuck	No	Yes
Cassandra-9549	CS3	Resource Leak	Slow	Yes	No
Cassandra-9486	CS4	Performance Bug	Slow	Yes	No
HDFS-8429	HF1	Uncaught Error	Stuck	Yes	Yes
HDFS-11377	HF2	Indefinite Blocking	Stuck	No	Yes
HDFS-11352	HF3	Deadlock	Stuck	Yes	No
HDFS-4233	HF4	Uncaught Error	Data Loss	Yes	No
HBase-18137	HB1	Infinite Loop	Stuck	Yes	No
HBase-16429	HB2	Deadlock	Stuck	Yes	No
HBase-21464	HB3	Logic Error	Stuck	Yes	No
HBase-21357	HB4	Uncaught Error	Denial	Yes	No
HBase-16081	HB5	Indefinite Blocking	Silent	Yes	No
MapReduce-6351	MR1	Deadlock	Stuck	Yes	No
MapReduce-6190	MR2	Infinite Loop	Stuck	Yes	No
MapReduce-6957	MR3	Improper Err Handling	Stuck	Yes	No
MapReduce-3634	MR4	Uncaught Error	Zombie	Yes	No
Yarn-4254	YN1	Improper Err Handling	Stuck	Yes	No

Detection Setup

Detector	Description
Client (**Panorama** [OSDI ’18])	instrument and monitor client responses
Probe (**Falcon** [SOSP ’11])	daemon thread in the process that periodically invokes internal functions with synthetic requests
Signal	script that scans logs and checks JMX metrics
Resource	daemon thread that monitors memory usage, disk and I/O health, and active thread count

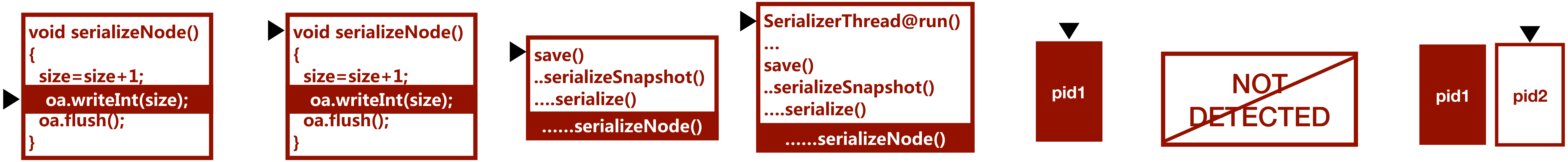
Baseline checkers

Detection Time

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Client	✗	2.47	2.27	✗	441	✗	✗	✗	✗	✗	✗	✗	✗	4.81	✗	6.62	✗	✗	✗	✗	8.54	7.38
Probe	✗	✗	✗	✗	15.84	✗	✗	✗	✗	✗	✗	✗	✗	4.71	✗	7.76	✗	✗	✗	✗	✗	✗
Signal	12.2	0.63	1.59	0.4	5.31	✗	✗	✗	✗	✗	✗	0.77	0.619	✗	0.62	61.0	✗	✗	✗	✗	0.60	1.16
Res.	5.33	0.56	0.72	17.17	209.5	✗	-19.65	✗	-3.13	✗	✗	0.83	✗	✗	✗	0.60	✗	✗	✗	✗	✗	✗
Watch	4.28	-5.89	3.00	41.19	-3.73	4.63	46.56	38.72	1.10	6.20	3.17	2.11	5.41	7.89	✗	0.80	5.89	1.01	4.07	1.46	4.68	✗

Detection times (in secs) for the real-world cases

Detection Localization



```
void serializeNode()
{
  size=size+1;
   oa.writeInt(size);
  oa.flush();
}
```

```
void serializeNode()
{
  size=size+1;
   oa.writeInt(size);
  oa.flush();
}
```

```
save()
..serializeSnapshot()
....serialize()
  ......serializeNode()
```

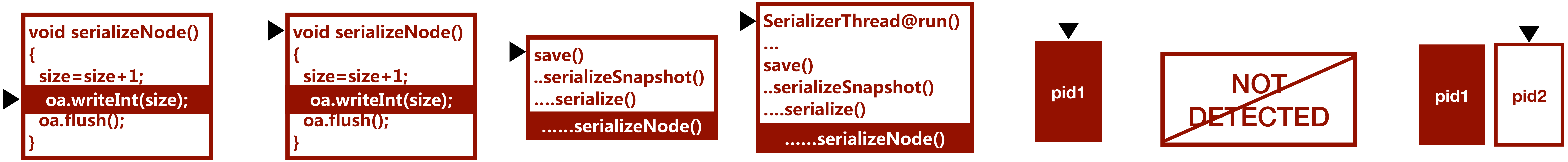
```
SerializerThread@run()
...
save()
..serializeSnapshot()
....serialize()
  ......serializeNode()
```

➼ faulty instr ✼ faulty func ✺ faulty call chain ◗ faulty entry ● faulty proc n/a no localize ❍ misjudged proc

Detection Localization

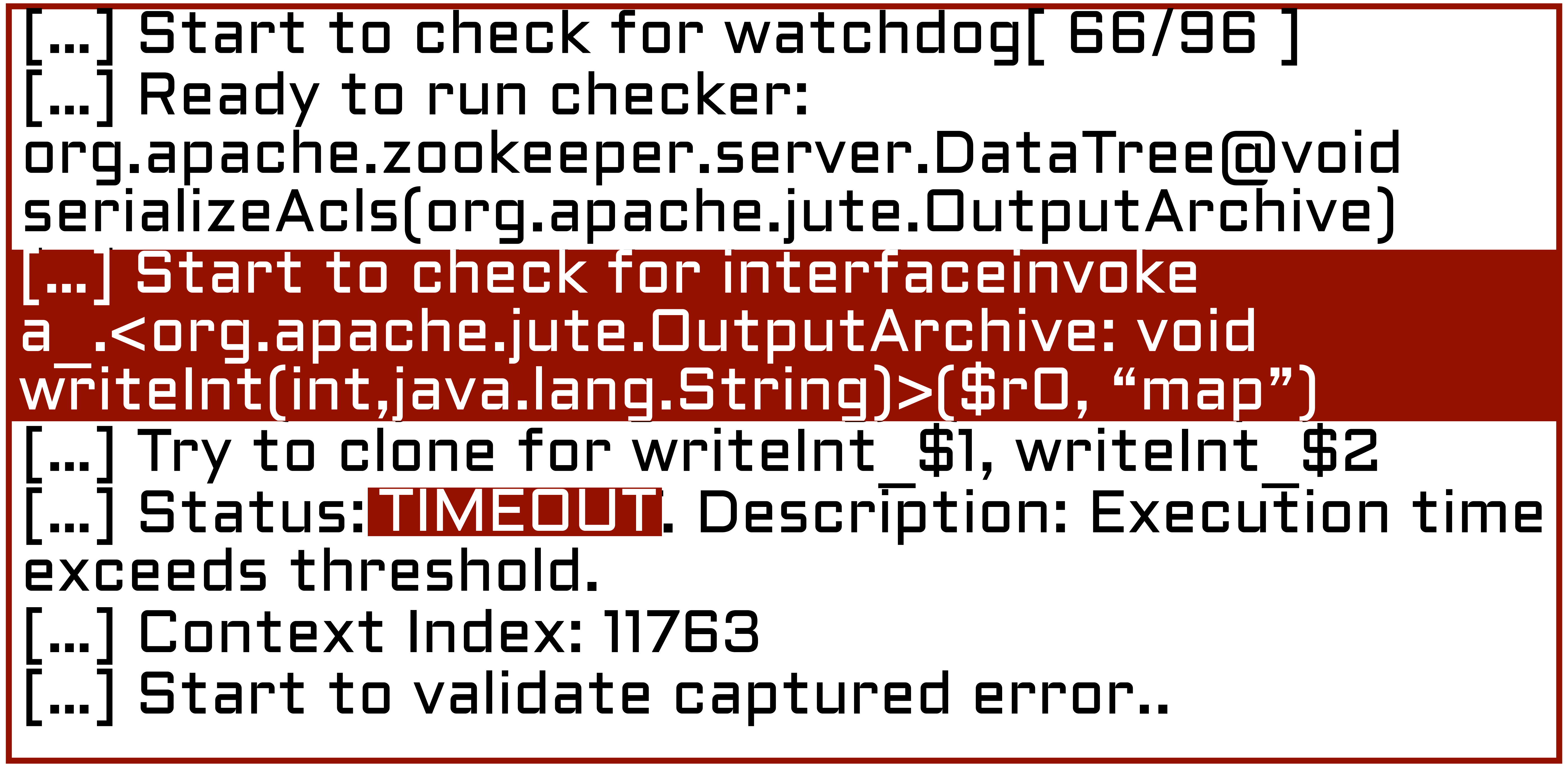
	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Client	n/a	●	●	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	●	n/a	❍	n/a	n/a	n/a	n/a	●	●
Probe	n/a	n/a	n/a	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	◗	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a
Signal	●	➸	●	●	➸	n/a	n/a	n/a	n/a	n/a	n/a	➸	➸	n/a	✺	✺	n/a	n/a	n/a	n/a	➸	➸
Res.	●	●	●	●	●	n/a	●	n/a	●	n/a	n/a	●	n/a	n/a	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a
Watch	➸	➸	●	✻	➸	✻	●	✻	✻	✺	➸	➸	➸	➸	n/a	➸	✺	➸	➸	✺	➸	n/a

Failure localization for the real-world cases.



➸ **faulty instr** ✻ **faulty func** ✺ **faulty call chain** ◗ **faulty entry** ● **faulty proc** **n/a no localize** ❍ **misjudged proc**

Discover new bug

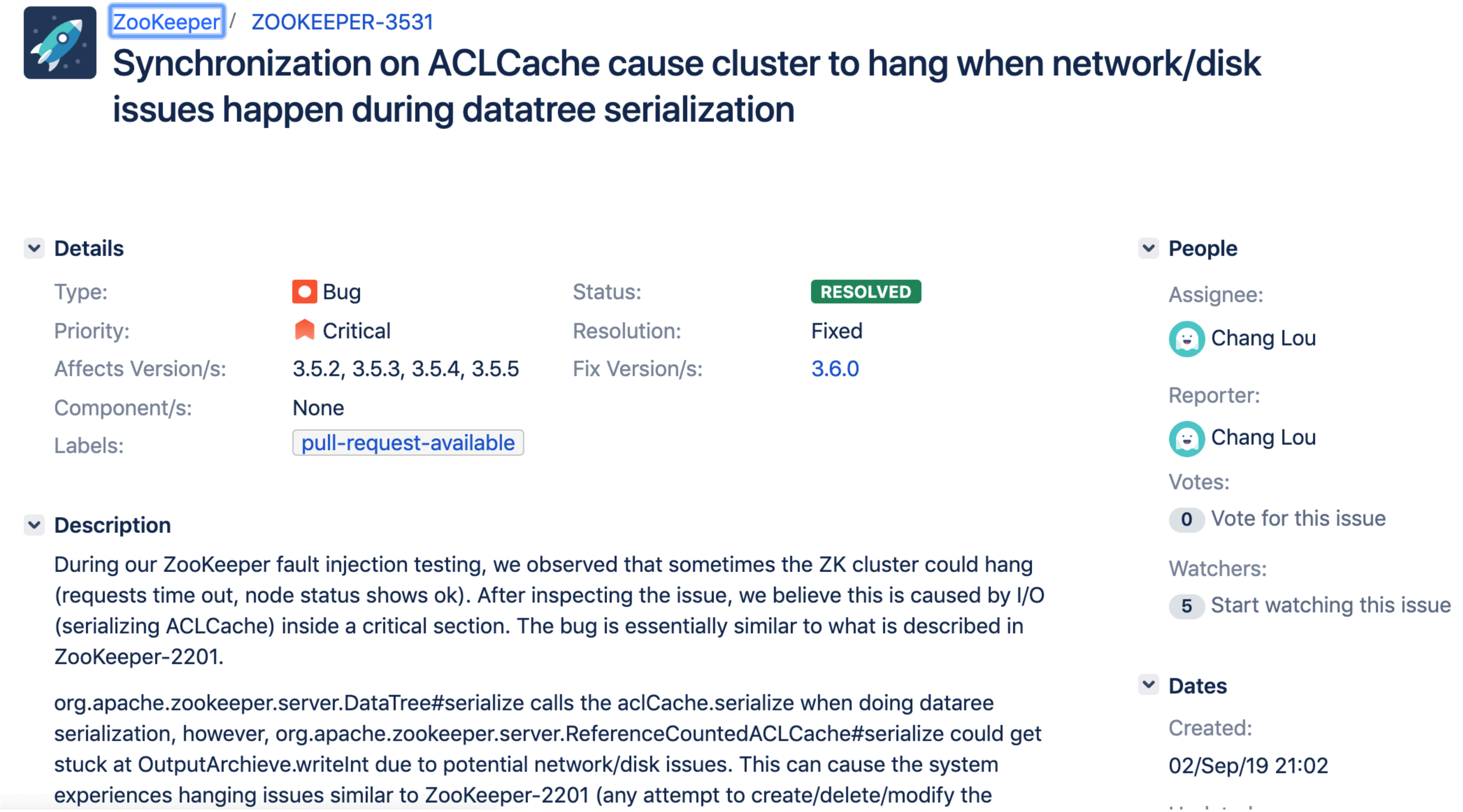


```
[...] Start to check for watchdog[ 66/96 ]
[...] Ready to run checker:
org.apache.zookeeper.server.DataTree@void
serializeAcls(org.apache.jute.OutputArchive)
[...] Start to check for interfaceinvoke
a_.<org.apache.jute.OutputArchive: void
writeInt(int,java.lang.String)>($r0, “map”)
[...] Try to clone for writeInt_$1, writeInt_$2
[...] Status: TIMEOUT. Description: Execution time
exceeds threshold.
[...] Context Index: 11763
[...] Start to validate captured error..
```

→

```
public void serializeAcls()
{
    ...
    synchronized(this)
    {
        oa.writeInt(longKeyMap.size(),   “map");
        for (Map.Entry<Long, List<ACL>> val :
        longKeyMap.entrySet()) {
          oa.writeLong(val.getKey(), "long");
        }
    }
}
```

Discover new bug



ZooKeeper / ZOOKEEPER-3531

Synchronization on ACLCache cause cluster to hang when network/disk issues happen during datatree serialization

Details

Type:	Bug	Status:	RESOLVED
Priority:	Critical	Resolution:	Fixed
Affects Version/s:	3.5.2, 3.5.3, 3.5.4, 3.5.5	Fix Version/s:	3.6.0
Component/s:	None		
Labels:	pull-request-available		

Description

During our ZooKeeper fault injection testing, we observed that sometimes the ZK cluster could hang (requests time out, node status shows ok). After inspecting the issue, we believe this is caused by I/O (serializing ACLCache) inside a critical section. The bug is essentially similar to what is described in ZooKeeper-2201.

org.apache.zookeeper.server.DataTree#serialize calls the aclCache.serialize when doing dataree serialization, however, org.apache.zookeeper.server.ReferenceCountedACLCache#serialize could get stuck at OutputArchieve.writeInt due to potential network/disk issues. This can cause the system experiences hanging issues similar to ZooKeeper-2201 (any attempt to create/delete/modify the

People

Assignee:

Chang Lou

Reporter:

Chang Lou

Votes:

0 Vote for this issue

Watchers:

5 Start watching this issue

Dates

Created:

02/Sep/19 21:02

False alarms

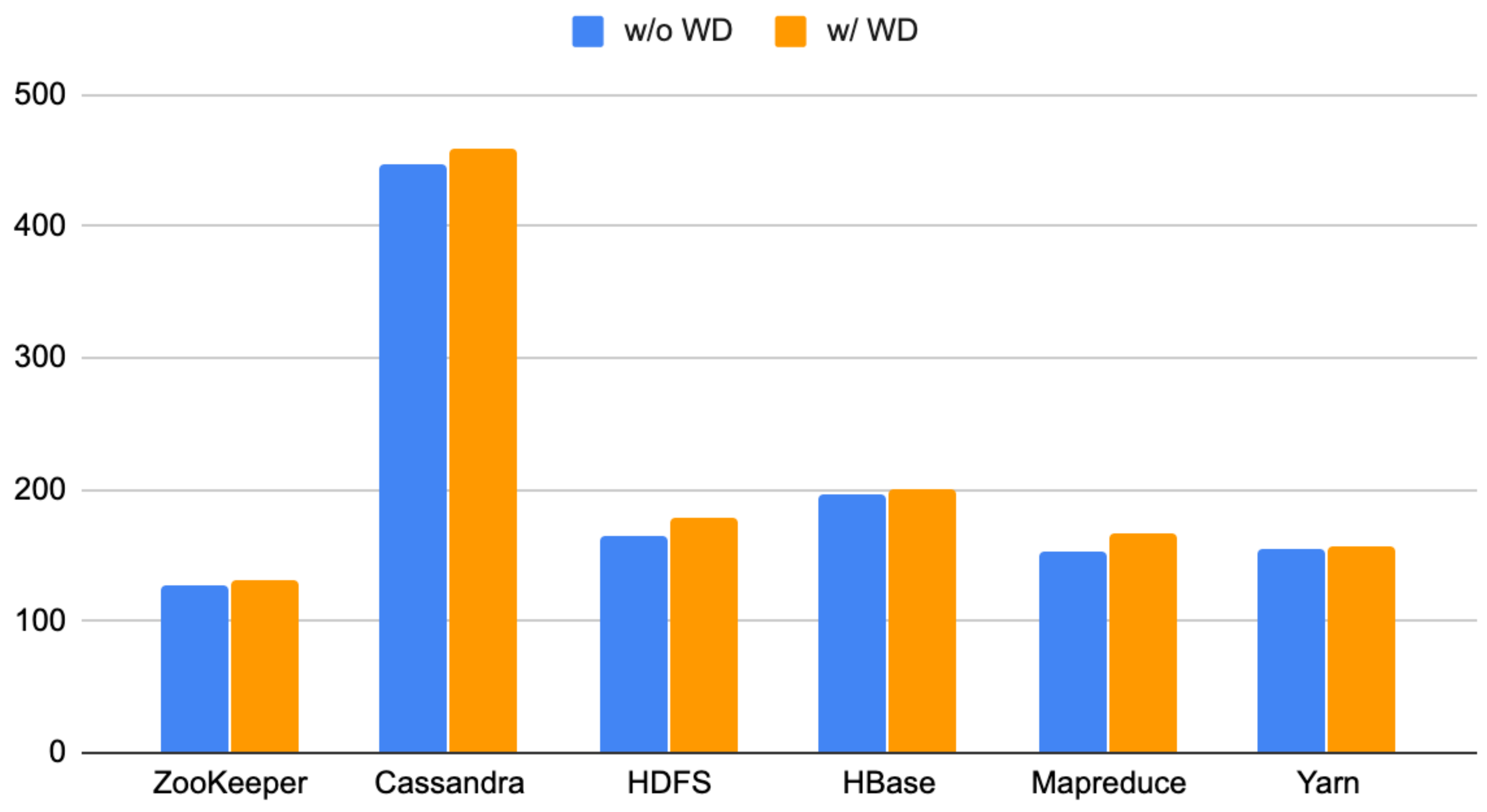
	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
probe	0%	0%	0%	0%	0%	0%
resource	0%-3.4%	0%-6.3%	0.05%-3.5%	0%-3.72%	0.33%-0.67%	0%-6.1%
signal	3.2%-9.6%	0%	0%-0.05%	0%-0.67%	0%	0%
watch.	0%-0.73%	0%-1.2%	0%	0%-0.39%	0%	0%-0.31%
watch_v.	0%-0.01%	0%	0%	0%-0.07%	0%	0%

False alarm ratios of all detectors for six systems under various setups

System overhead

	ZooKeeper	Cassandra	HDFS	HBase	MapReduce	Yarn
Base	428.0	3174.9	90.6	387.1	45.0	45.0
w/ Probe.	417.6	3128.2	89.4	374.3	44.9	44.9
w/ Resource.	424.8	3145.4	89.9	385.6	44.9	44.6
w/ Watch.	399.8	3014.7	85.1	366.4	42.1	42.3

Throughput (op/s) w/ different checkers



Heap memory usage w/ and w/o watchdogs

<u>5.0%-6.6% throughput overhead w/ watchdog</u>

<u>4.3% (avg) memory overhead w/ watchdog</u>

Conclusions

- **Modern software are increasingly complex and often fail partially**
  - these partial failures cannot be detected by process-level failure detectors
- **We conducted a study on 100 partial failure cases**
- **OmegaGen: a static analysis tool that automatically generates customized checkers**
  - successfully generate checkers for six systems and checkers can detect && localize 18/22 real-world partial failures
  - watchdog report helps to quickly discover a new bug in the latest zookeeper

Related Work

- **Partial failures**
  - Fail-Stutter [HotOS ’01], IRON [SOSP’05], Limplock [SoCC '13], Fail-Slow Hardware [FAST '18], Gray Failure [HotOS '17]
- **Failure detection**
  - Gossip [Middleware '98], ϕ [SRDS '04], Falcon [SOSP '11], Pigeon [NSDI ’13], Panorama [OSDI’18]
- **Software invariant generation**
  - Daikon [ICSE’99], InvGen [CAV’09], PCHECK [OSDI’16]