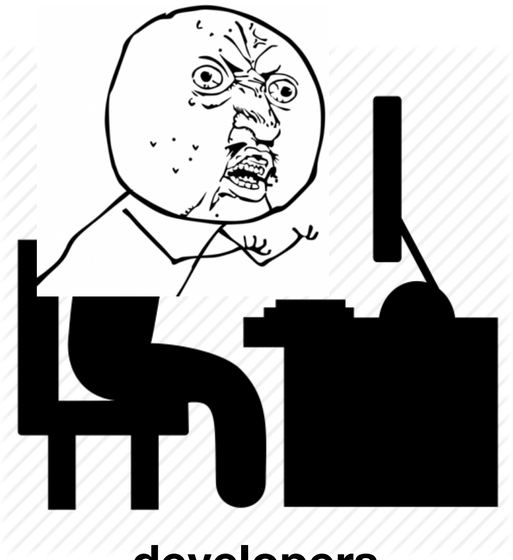
Comprehensive and Efficient Runtime Checking in System Software through Watchdogs

Chang Lou, Peng Huang, Scott Smith

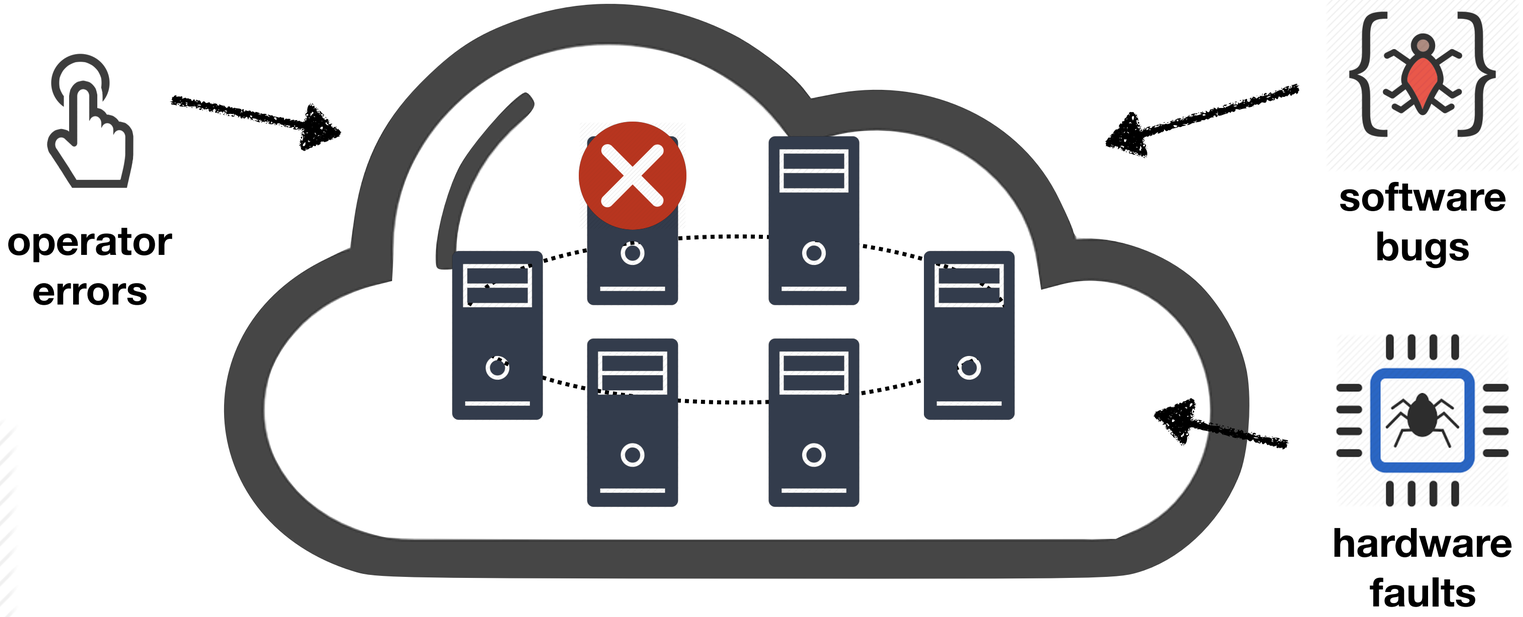
JOHNS HOPKINS
UNIVERSITY

HOTOS XVII

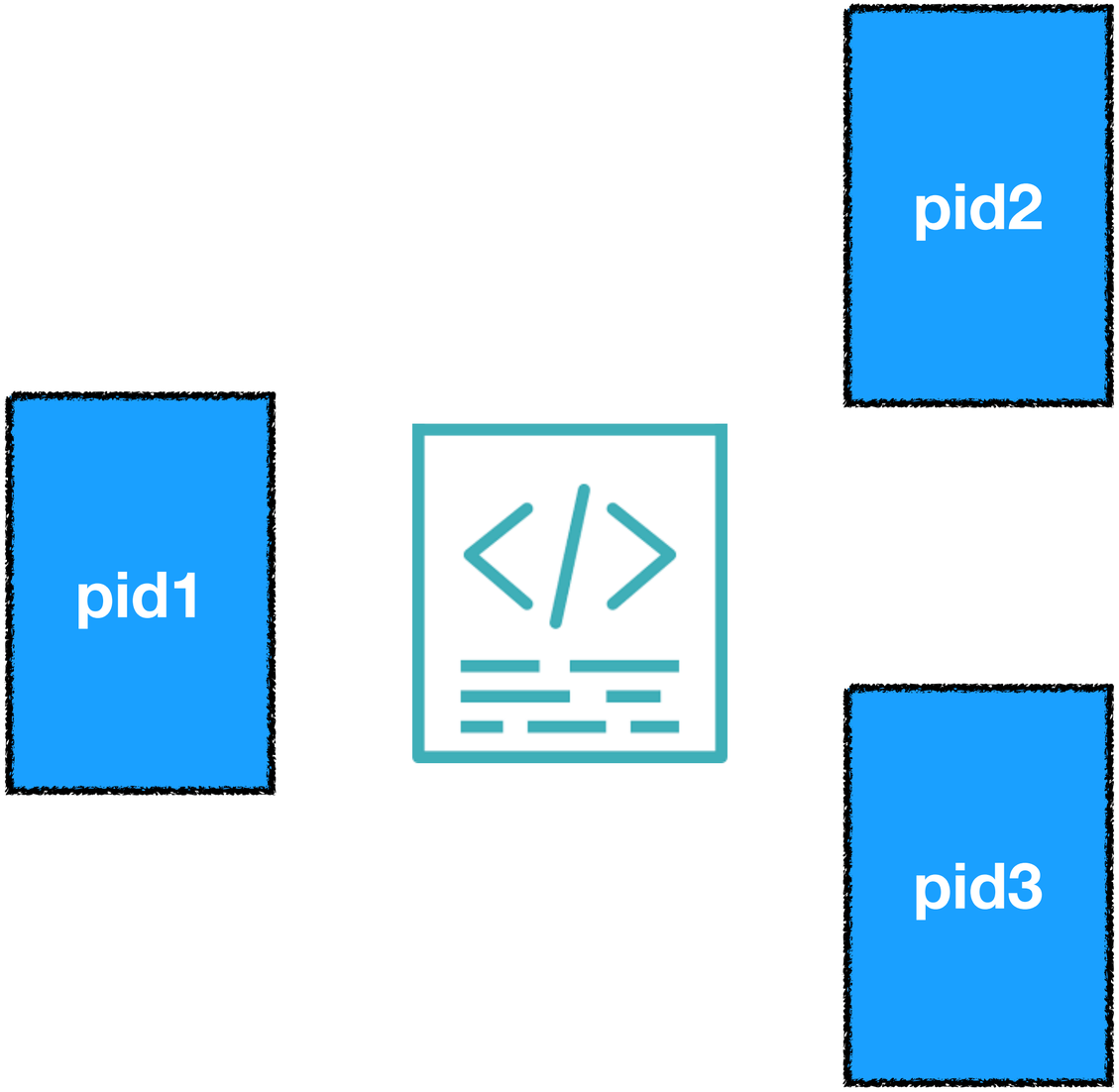
All large systems inevitably fail



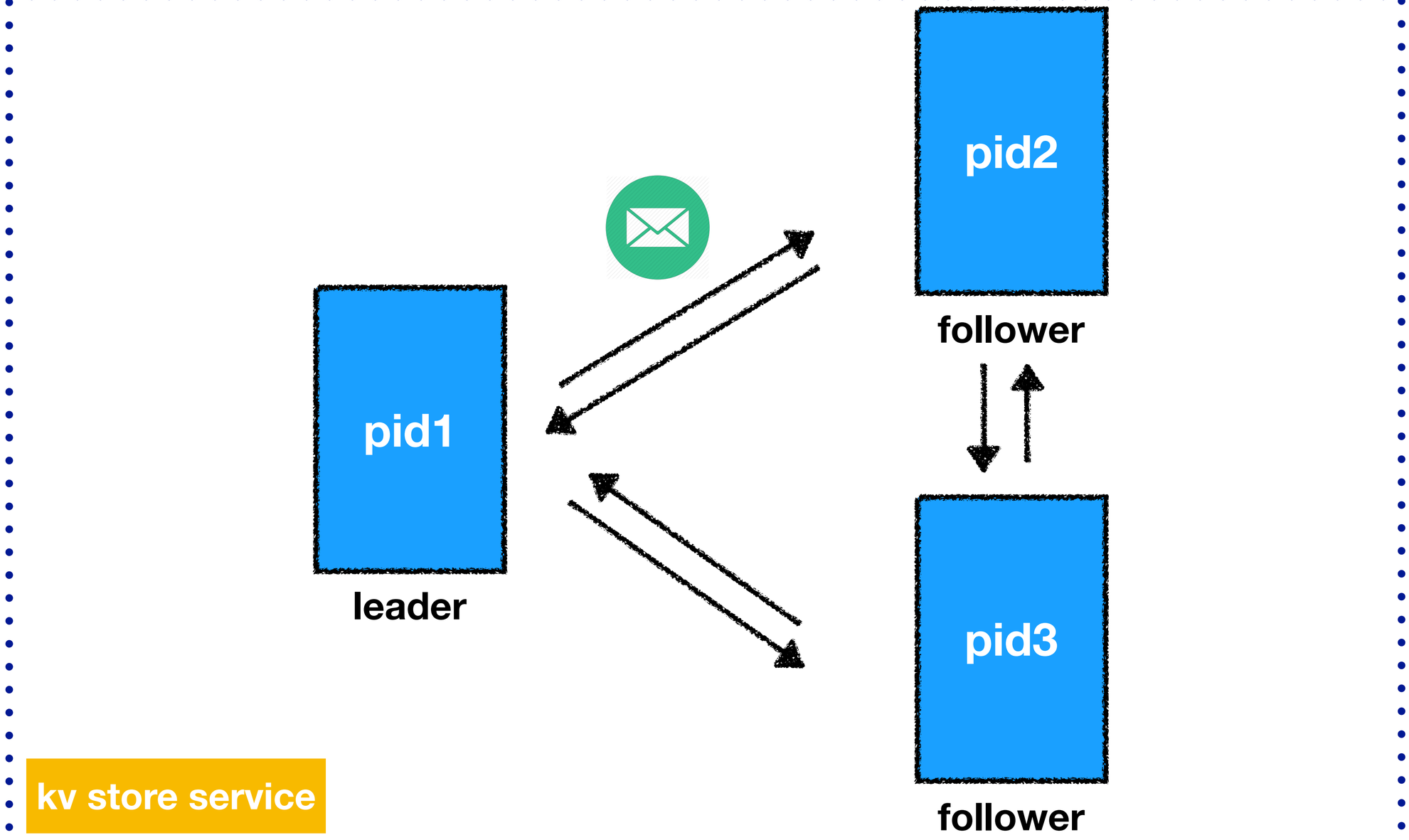
developers



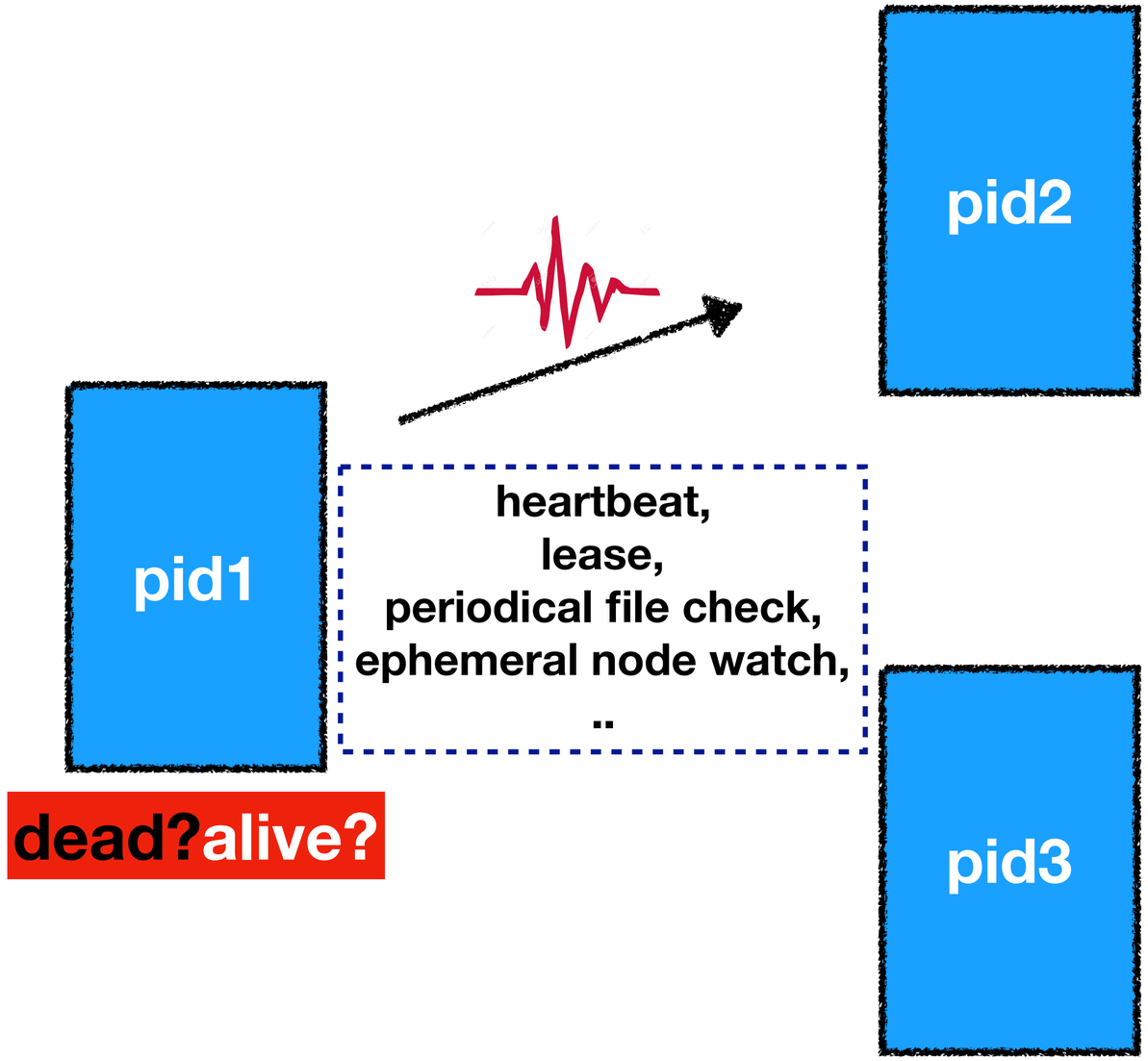
Process-level failure detector abstraction



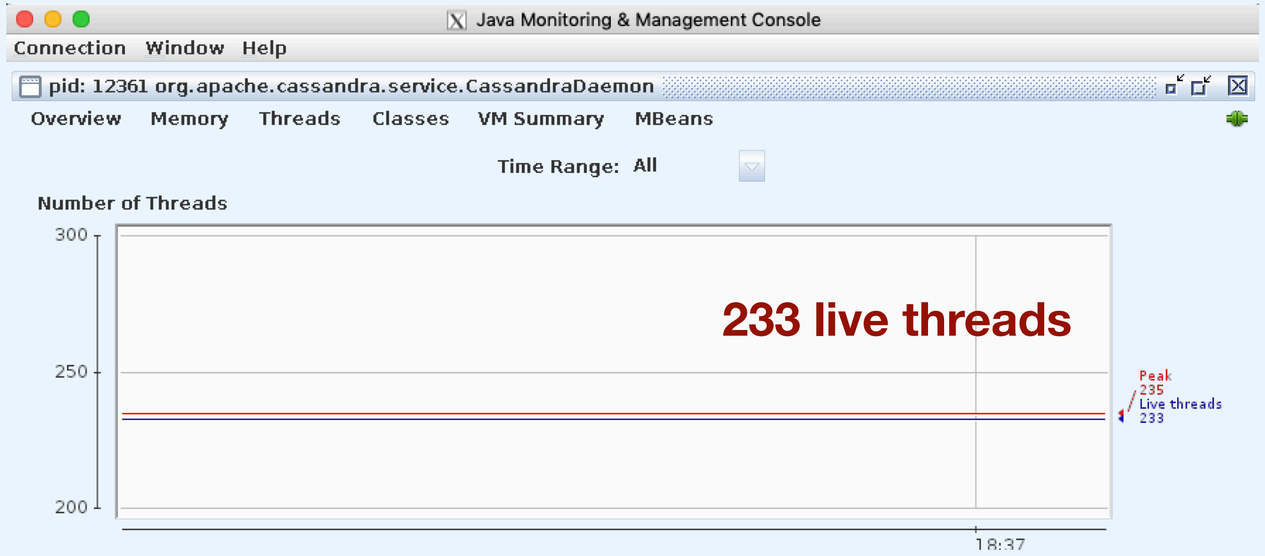
Process-level failure detector abstraction



Process-level failure detector abstraction



Modern software is complex



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

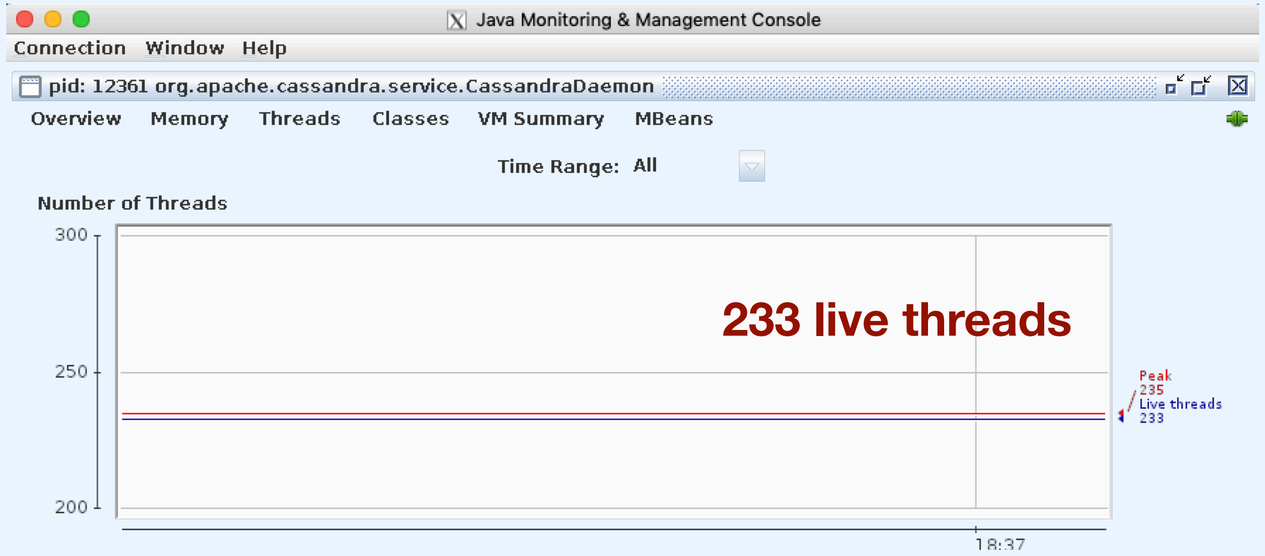
Local database operators

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



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

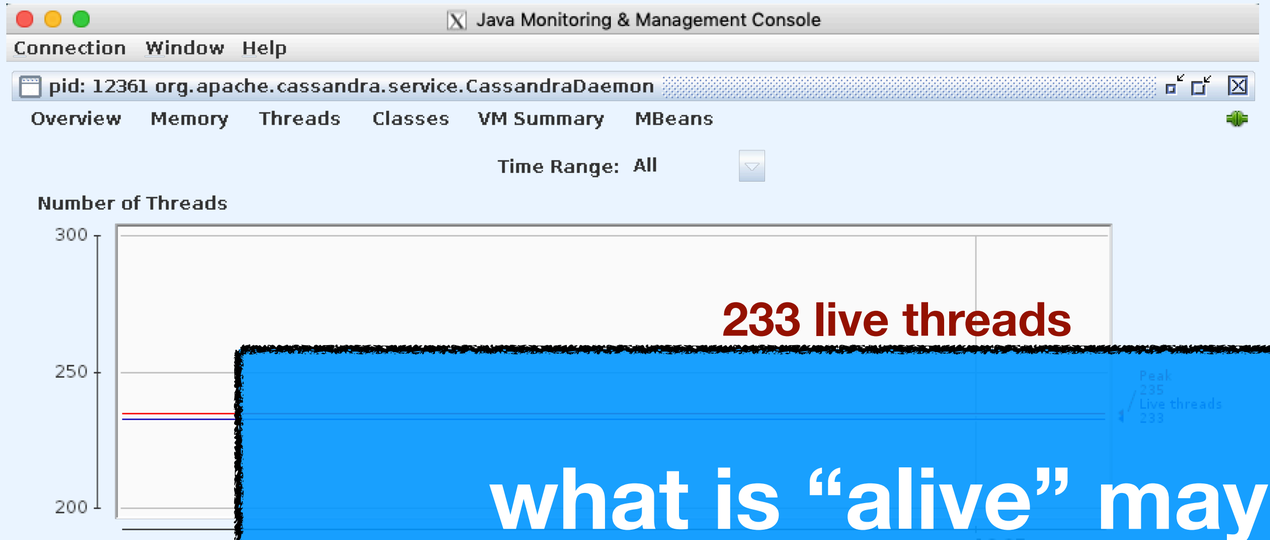
Local database operators

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



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
New I/O worker #1
Thread-7
New I/O worker #2

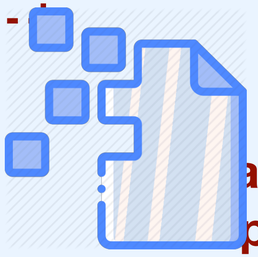
I/O workers

what is “alive” may no longer live

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

ackground 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

911 emergency services go down across the US after CenturyLink outage

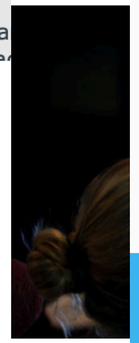
The Cloud
many issues

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

Zack Wl

Rackspace C

Load Bala



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

By

2018

HE COM

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

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

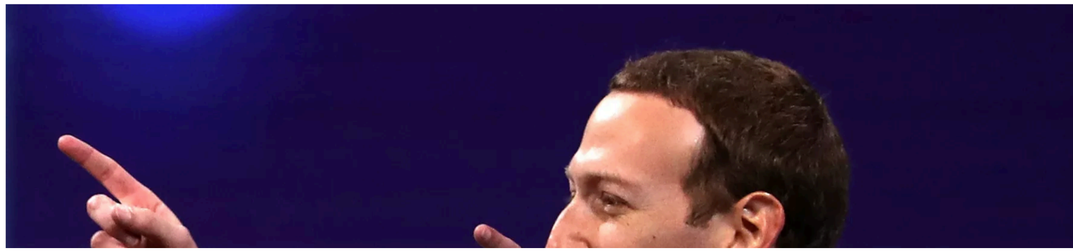
LEE HUTCHIN

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

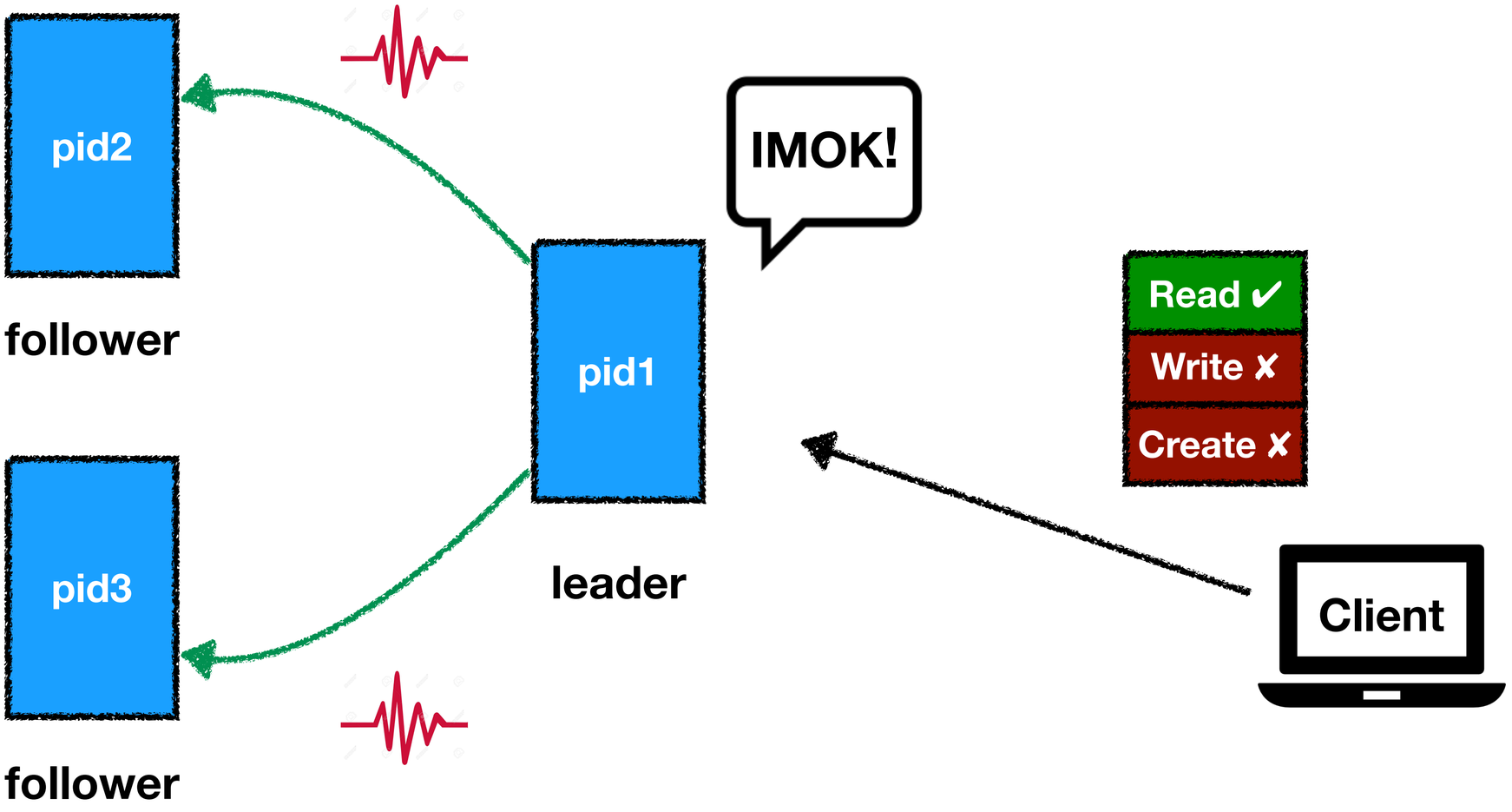
SHARE



Outline

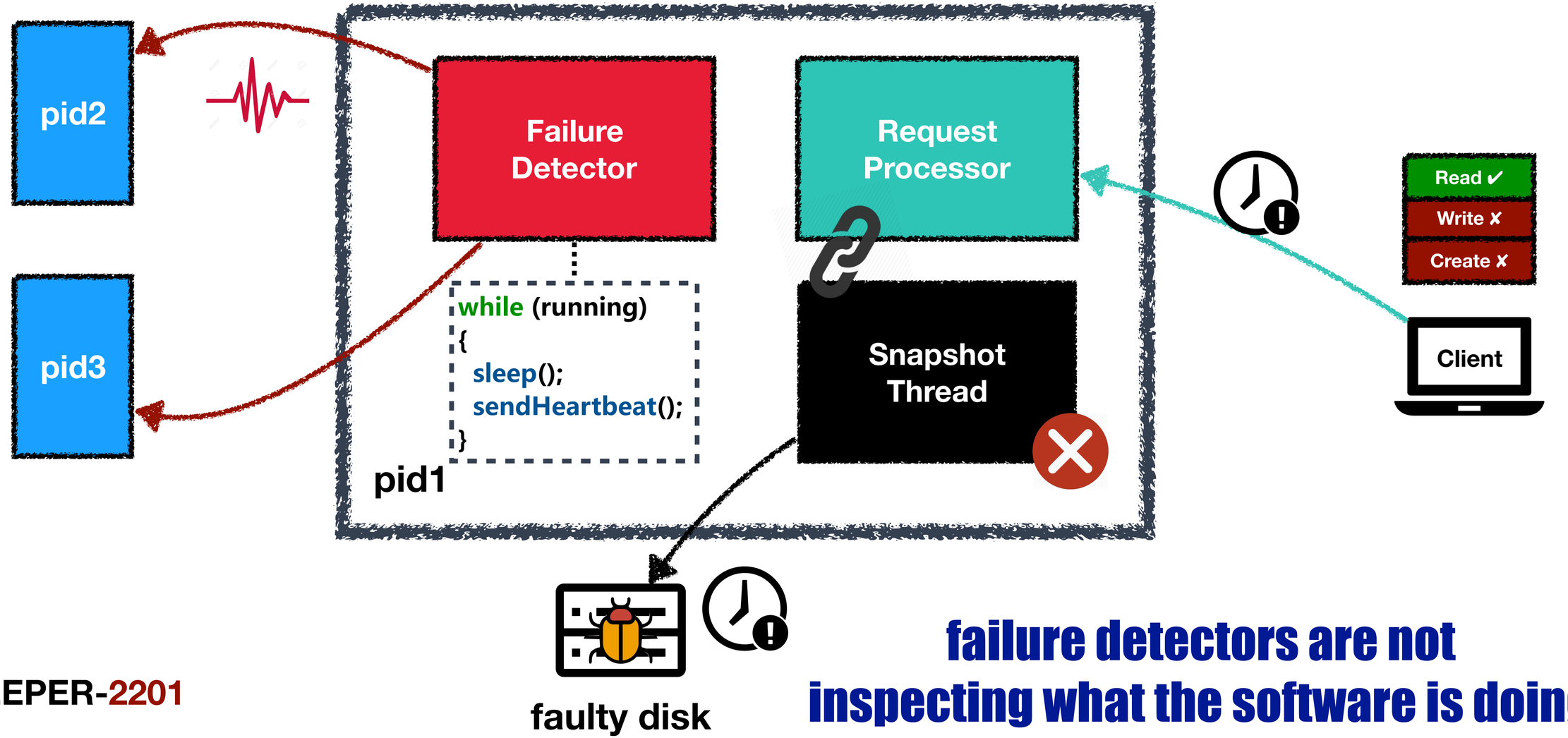
- Motivation
- **Intrinsic software watchdog abstraction**
  - hardware & software watchdogs
  - characteristics
  - checker approach
- **AutoWatchdog: a tool to generate watchdogs**
  - technique: program reduction
- **Challenges & Opportunities**

Why existing failure detectors cannot detect partial failures?



ZOOKEEPER-2201

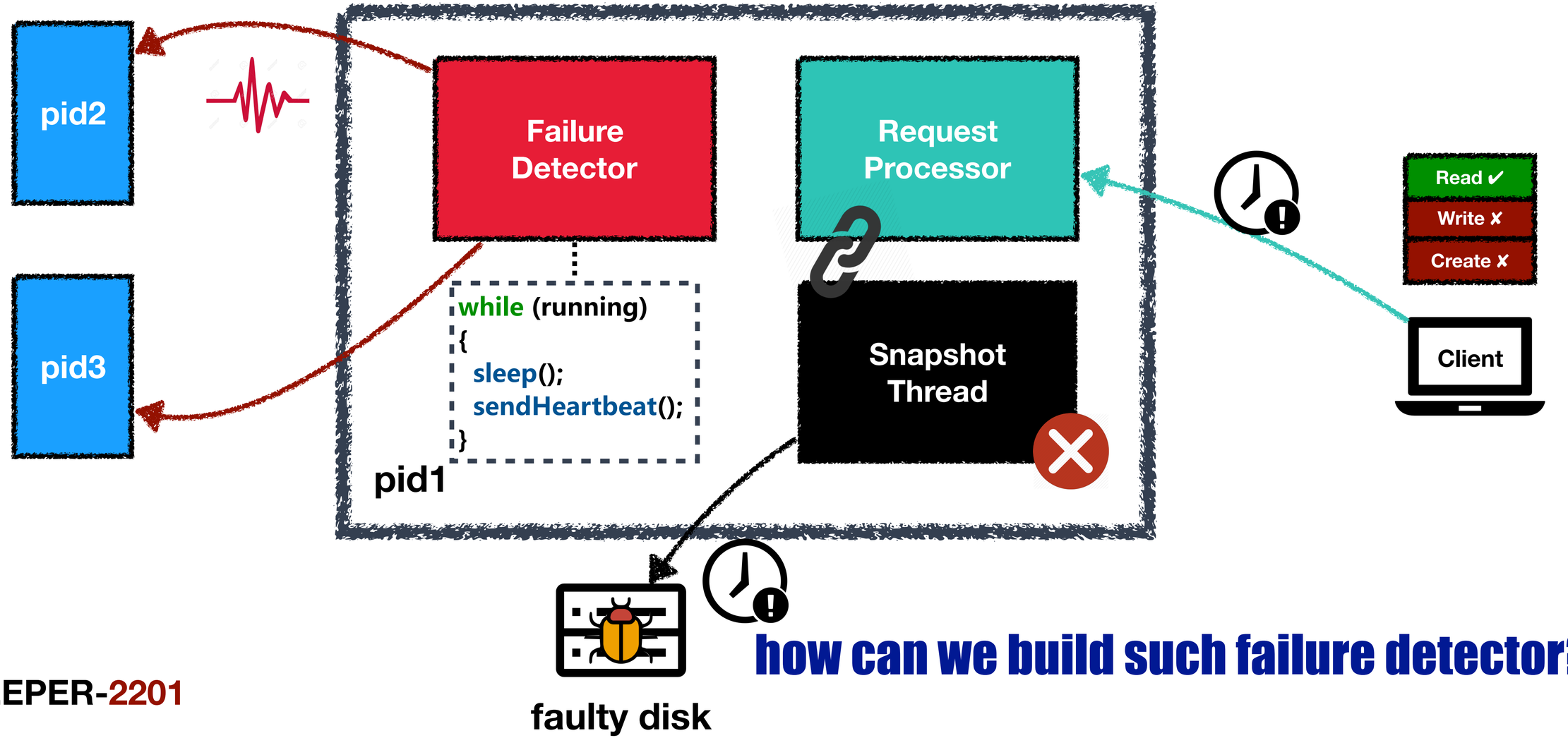
Why existing failure detectors cannot detect partial failures?



ZOOKEEPER-2201

failure detectors are not inspecting what the software is doing

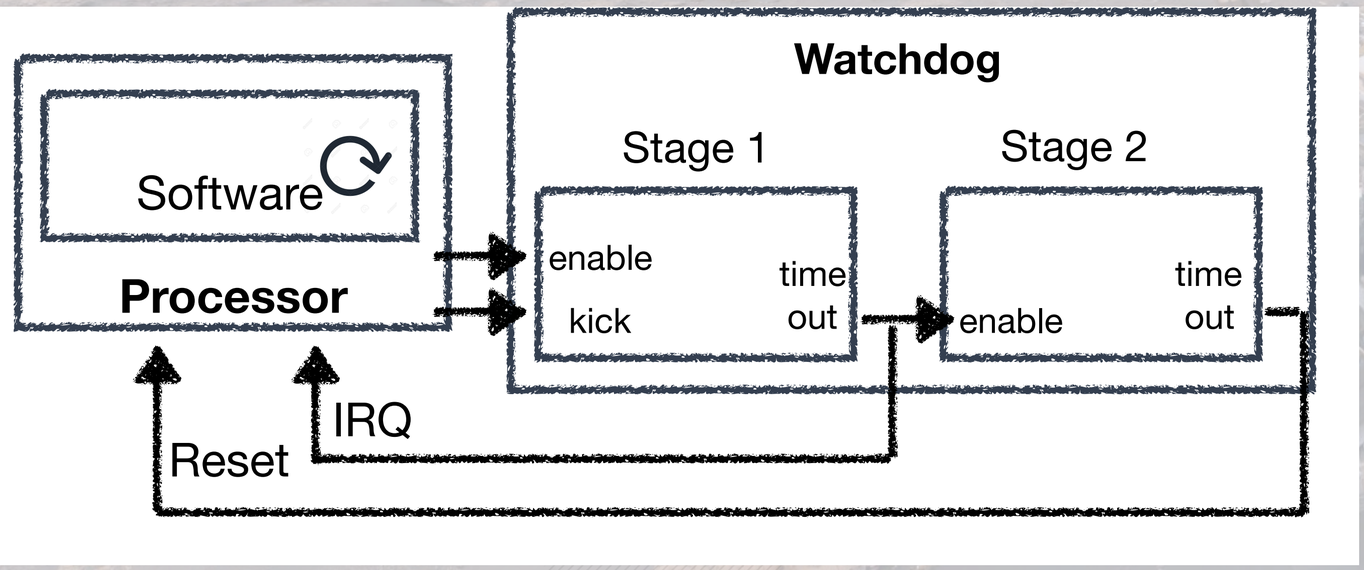
Lesson: failure detectors must comprehensively reflect the process internal status



how can we build such failure detector?

ZOOKEEPER-2201

Wisdom from embedded system: Watchdog



sanity checks

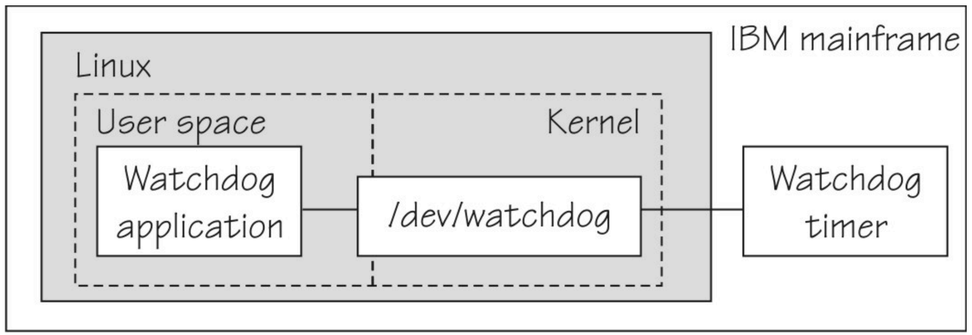
stack depth
status of some mechanical component
deadlock
…

hardware
mechanical component issues
…

software
infinite loop, accidental jump, deadlock…

environment
bit flip due to cosmic ray
…

Current software watchdogs are shallow

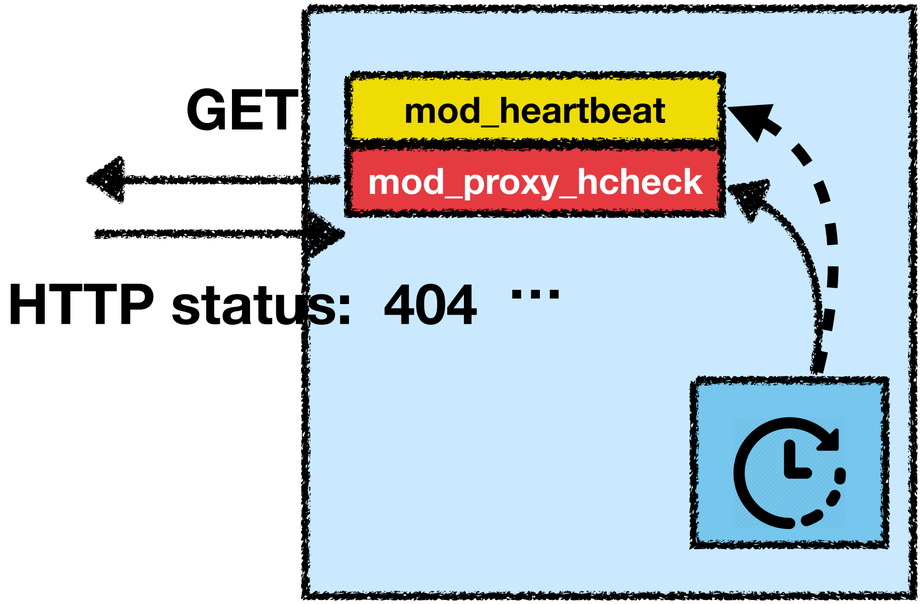


process table not full ✔

enough memory ✘

file table overflow ✘

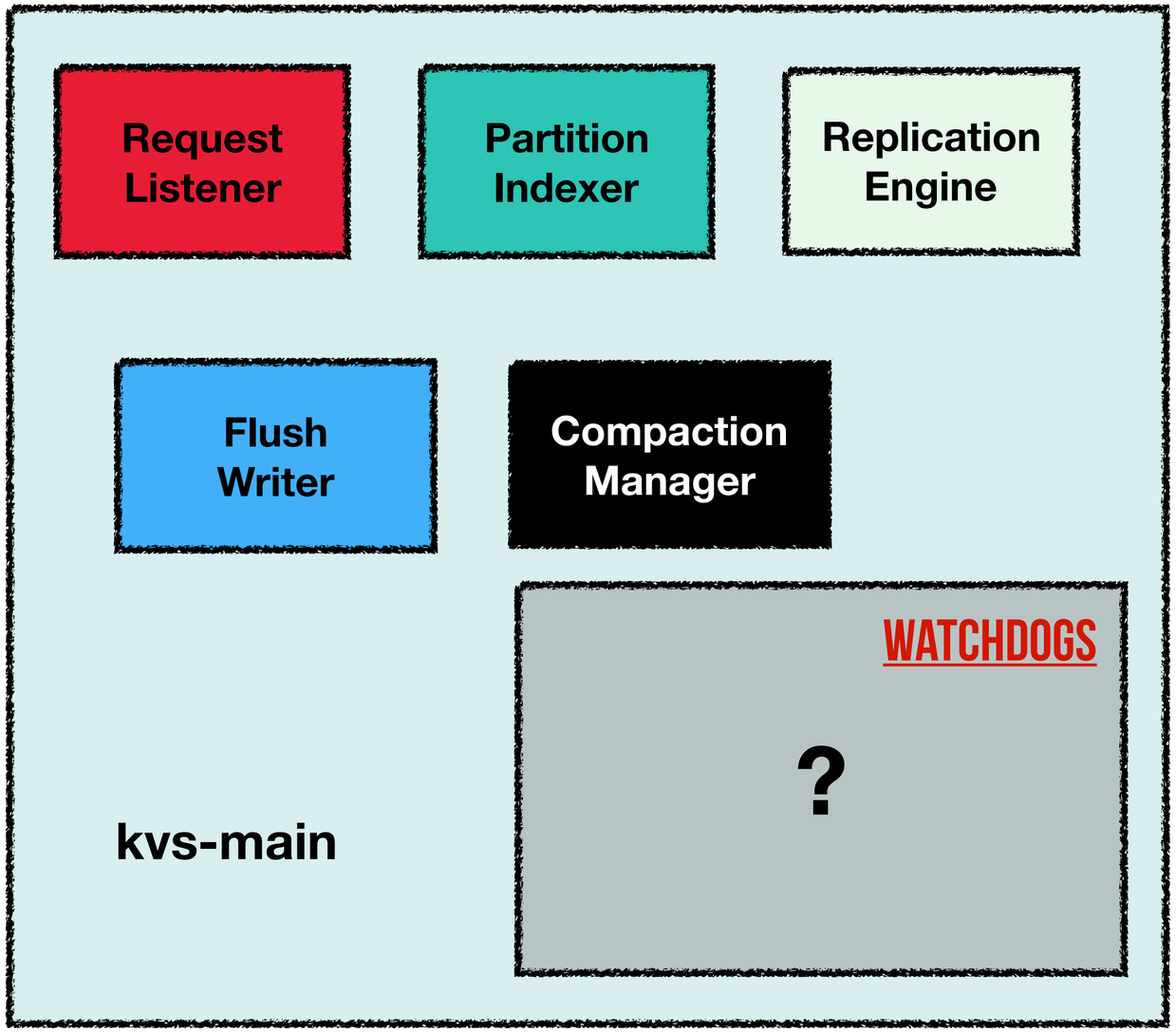
linux watchdog



httpd mod_watchdog

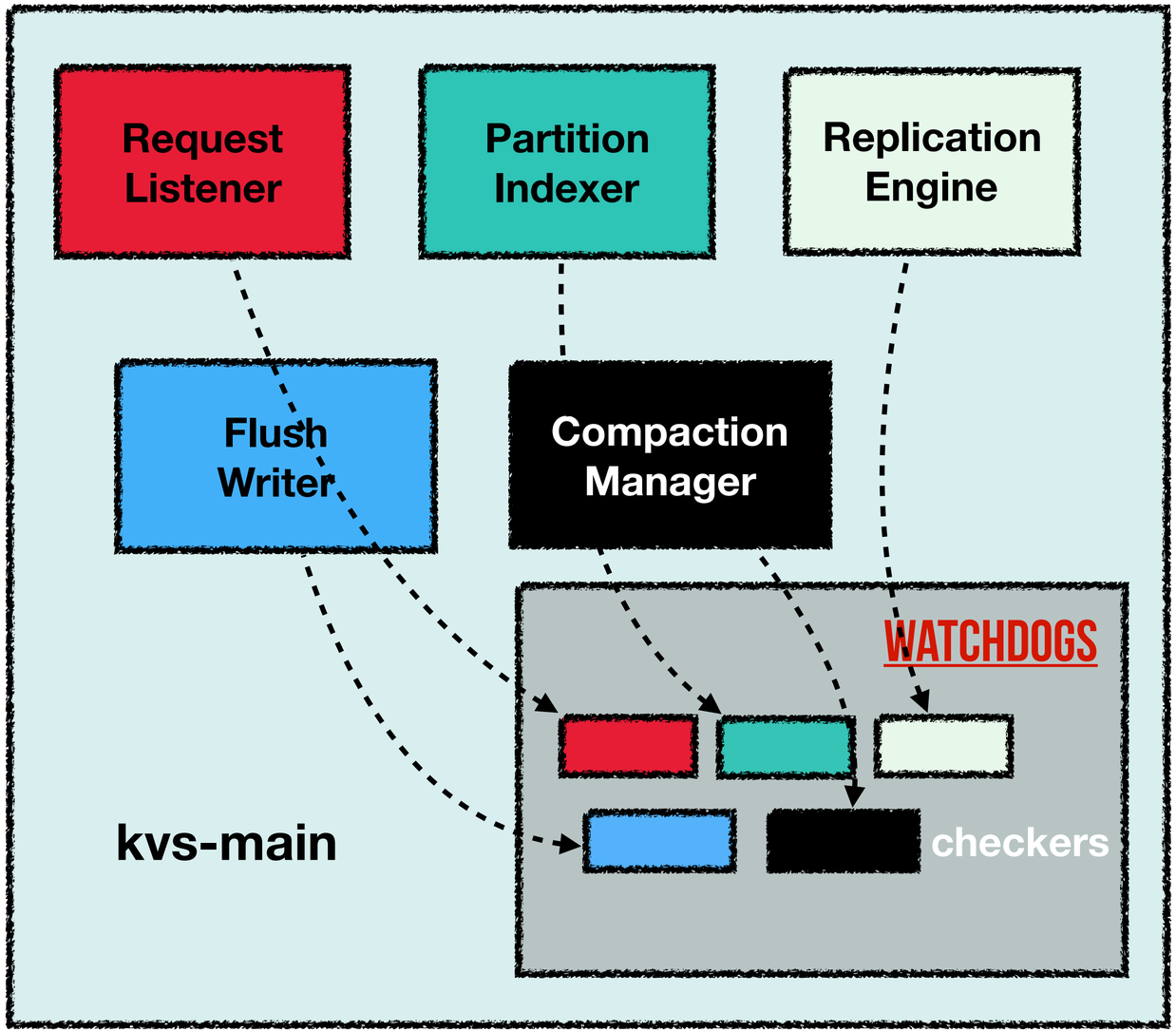
they are essentially equivalent to a crash failure detector + kill policy

Our solution: intrinsic software watchdog



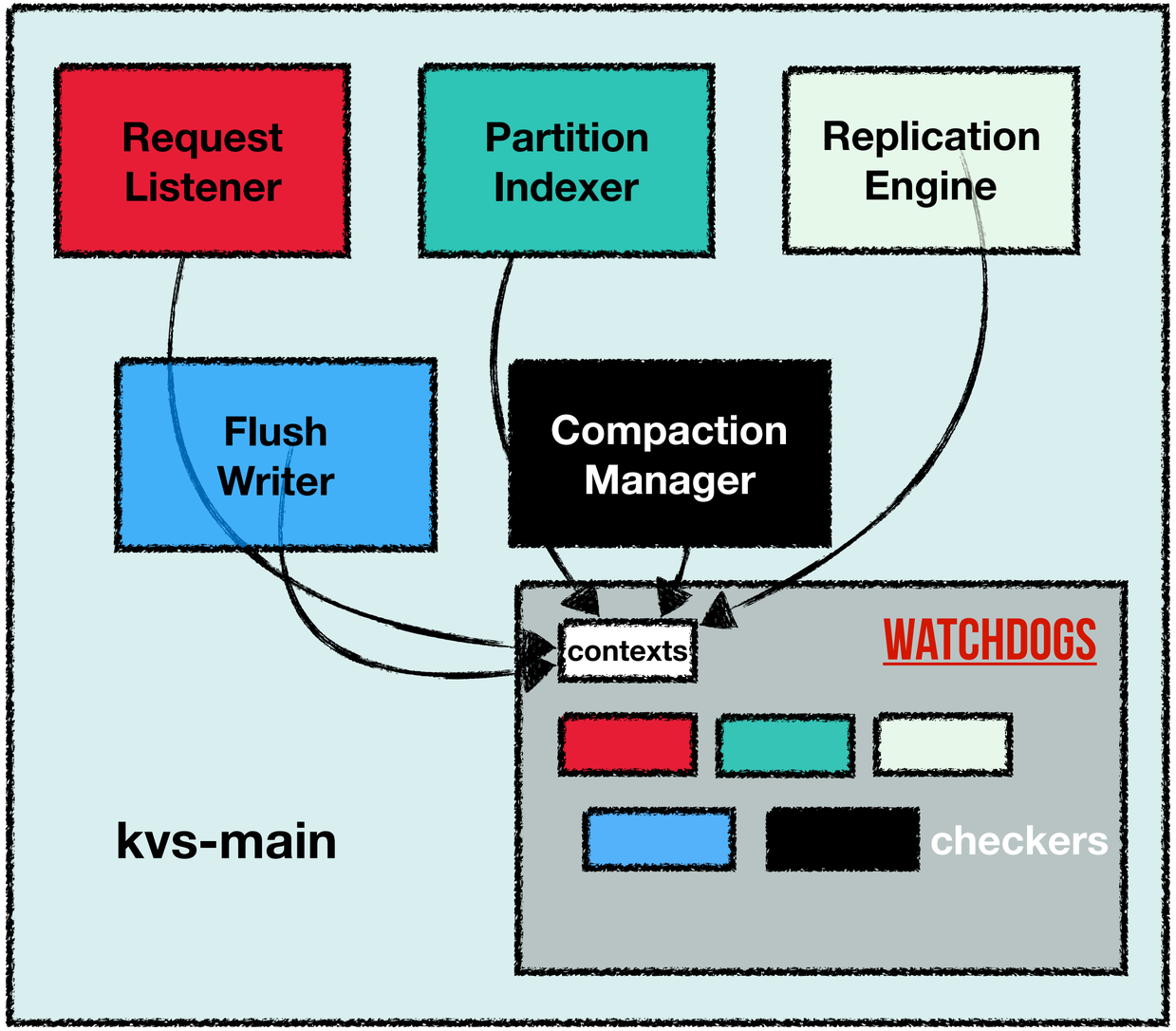
- What should intrinsic software watchdogs look like?

Intrinsic software watchdog



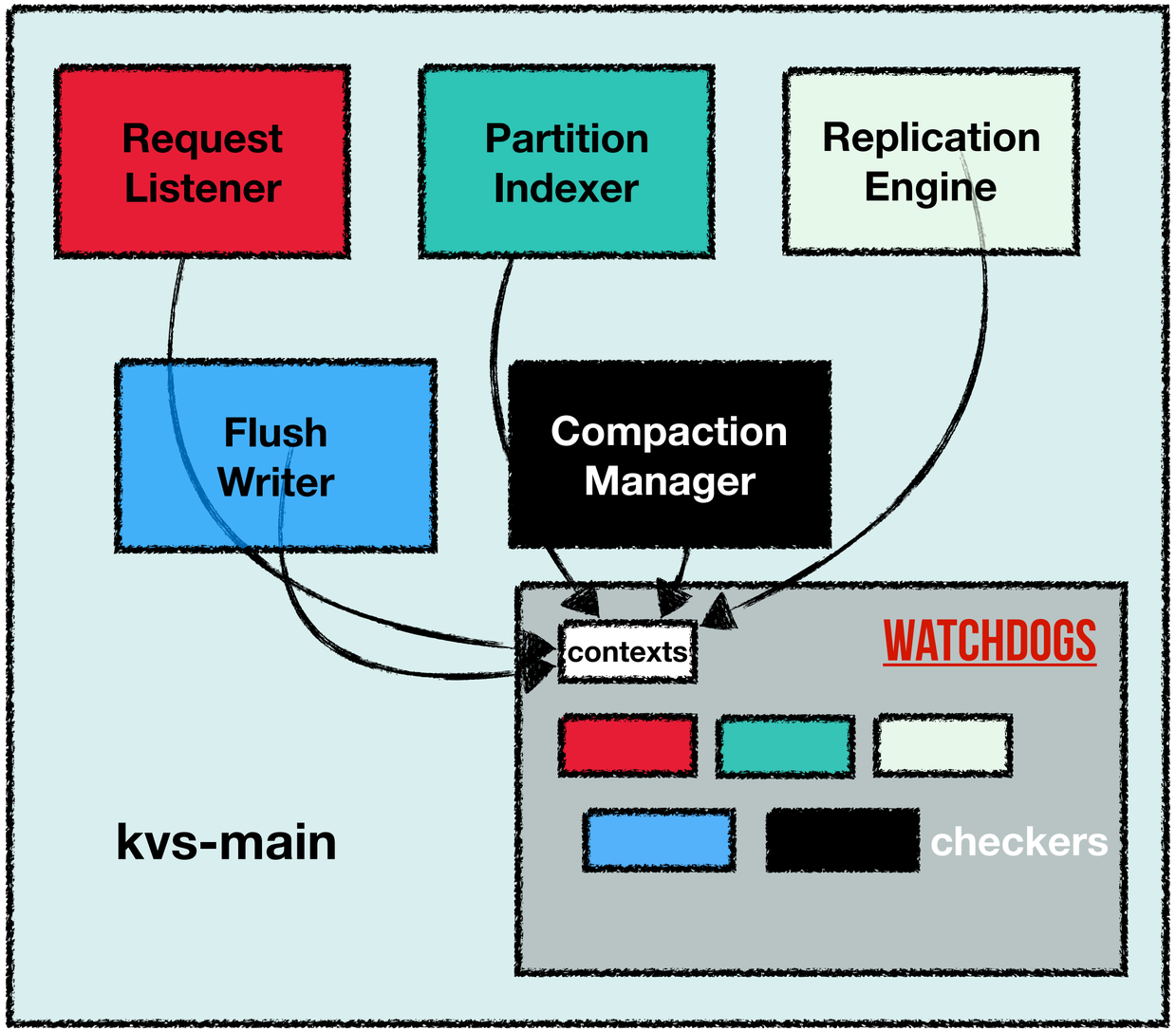
- **Design choice #1 Tailored checker**
  - checkers' logics are customized based on main execution logic

Intrinsic software watchdog



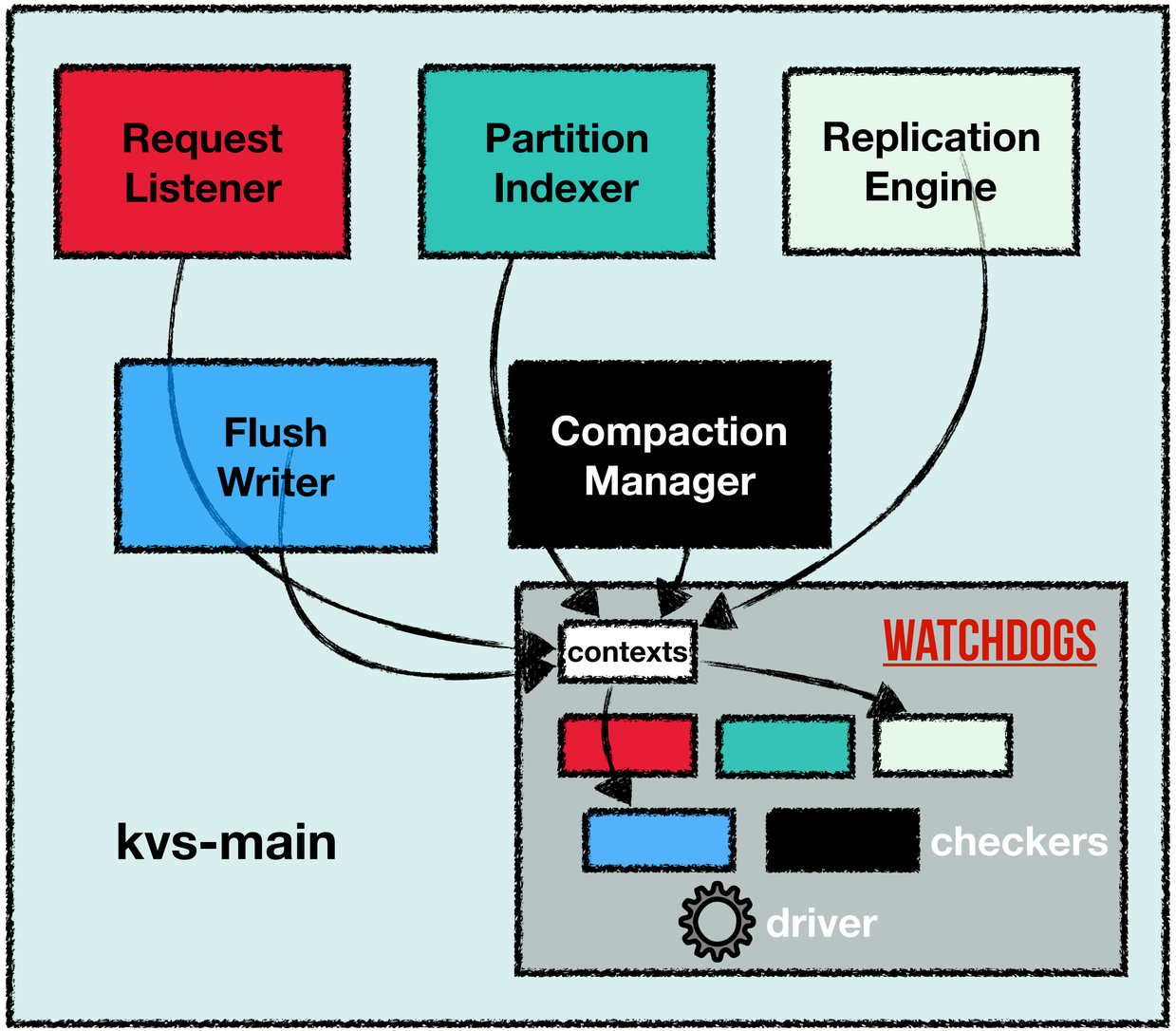
- **Design choice #2 Stateful checker**
  - checkers should faithfully reflect checked target status, which unavoidably requires collecting program states (contexts) from main execution

Intrinsic software watchdog



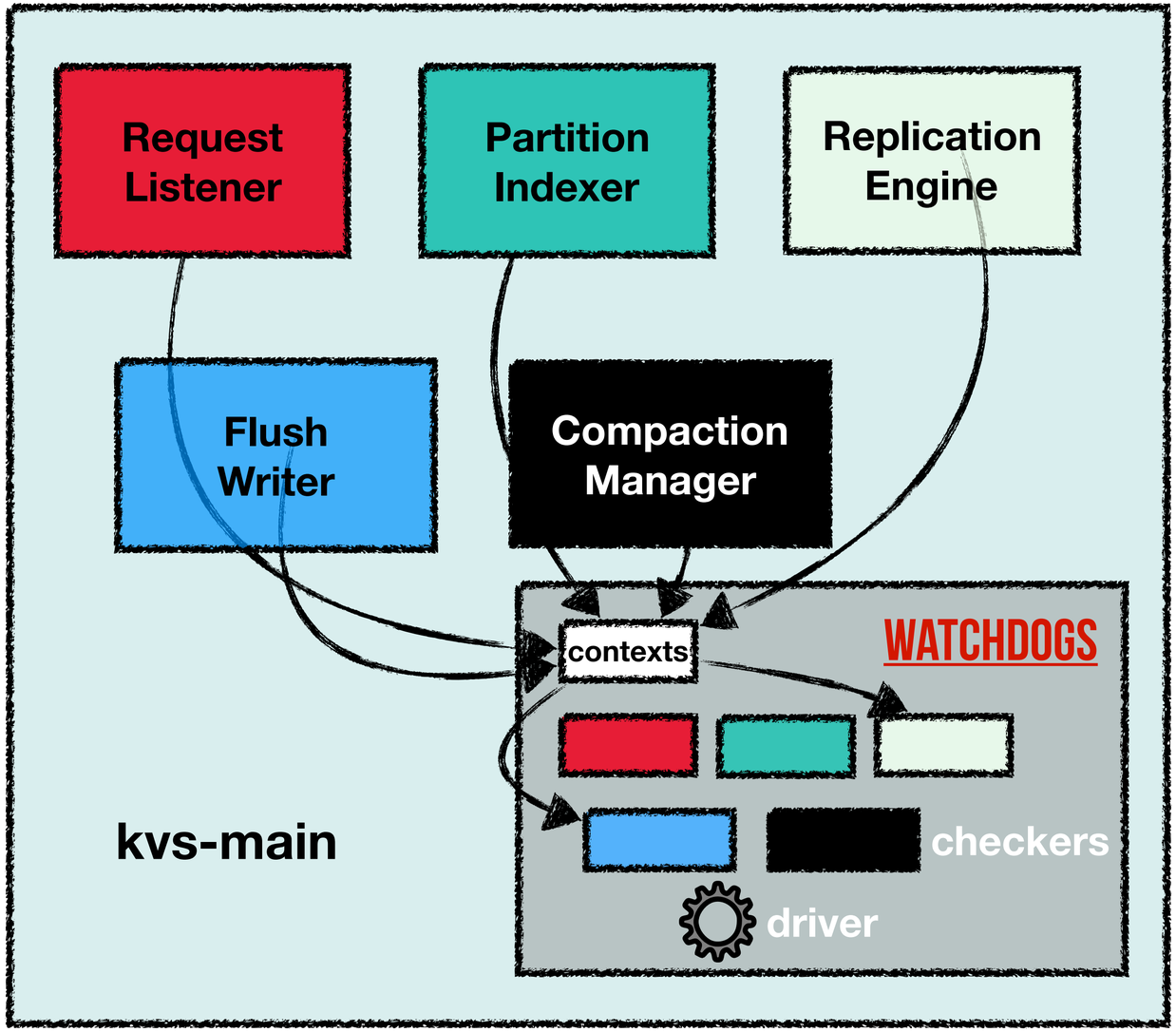
- **but with so many customized and stateful checkers, developers have to worry about two things:**
  - paying performance penalty in the normal execution even when there is no failure
  - checkers might introduce side effects or alter main execution

Intrinsic software watchdog



- **Design choice #3 Concurrent execution**
  - ✦ checkers run async with main execution
  - ✦ one-way state synchronization to provide isolation

Proposed intrinsic watchdog abstraction



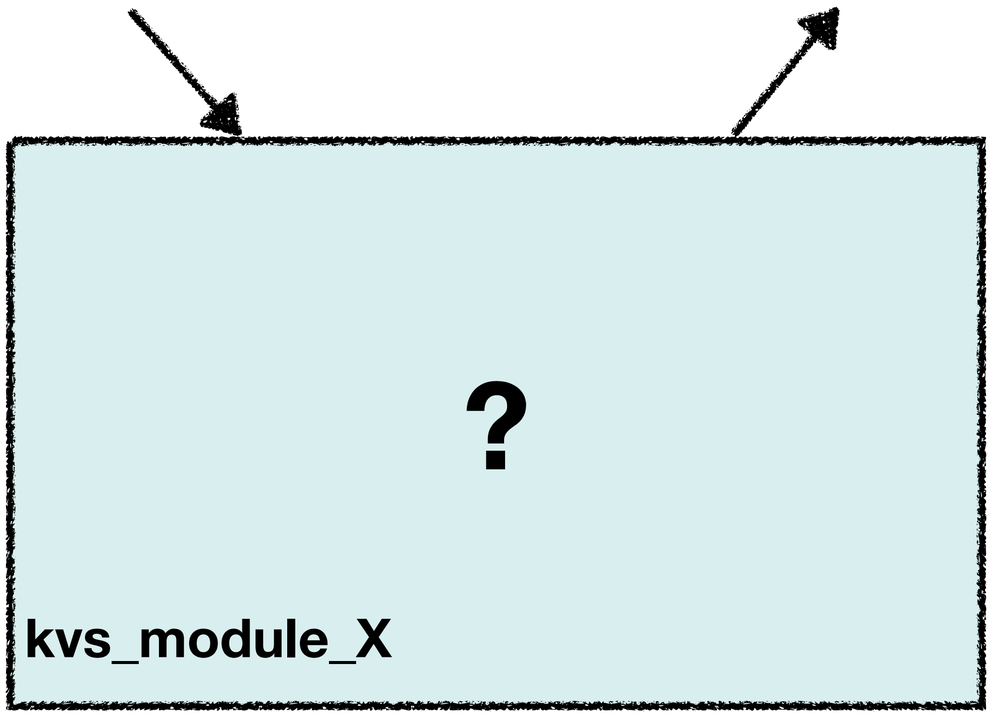
- **Checkers**
  - ✦ encapsulated checking procedures
- **Context Manager**
  - ✦ synchronize and manage states for checkers
- **Driver**
  - ✦ manage checker scheduling and execution

how can we construct watchdog checkers?

Try #1: Probing

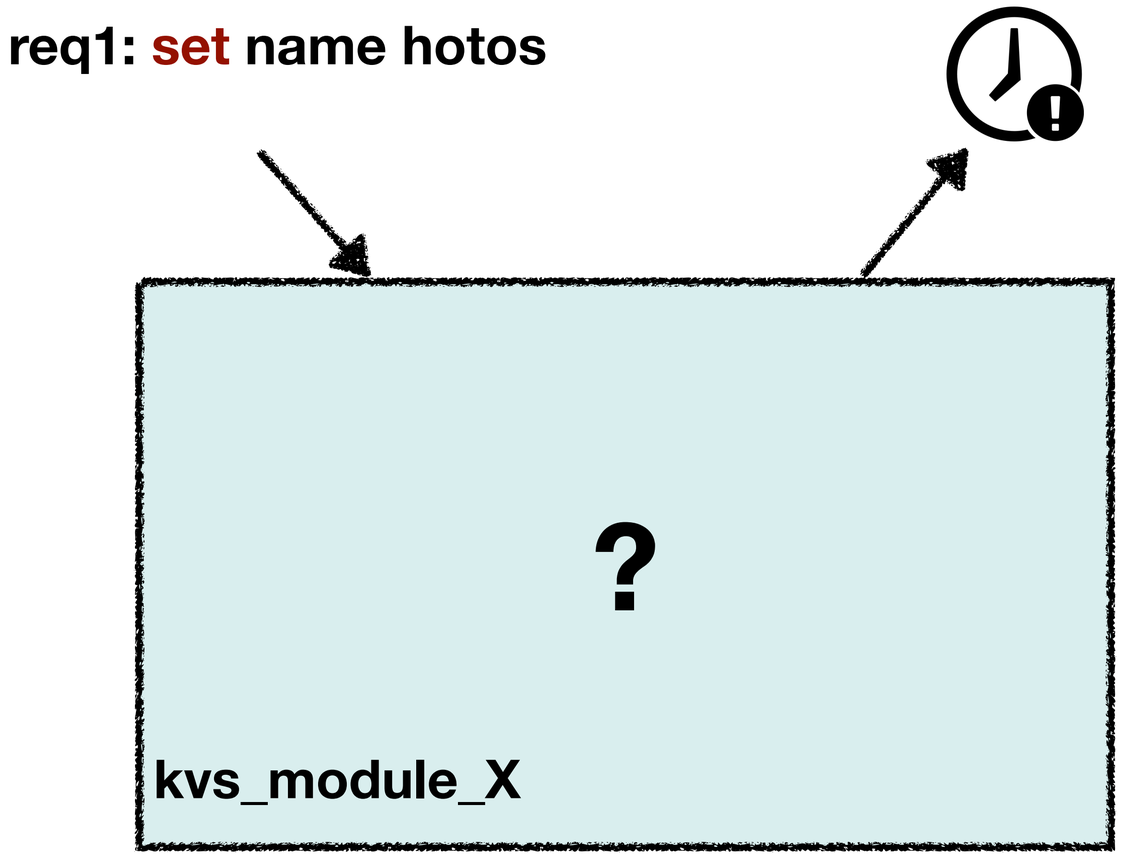
req1: set name hotos
req2: get name

resp1: “OK”
resp2: “hotos”



- periodically invoke some APIs with synthetic input and check
- perfect accuracy
  - ✦ no false alarm

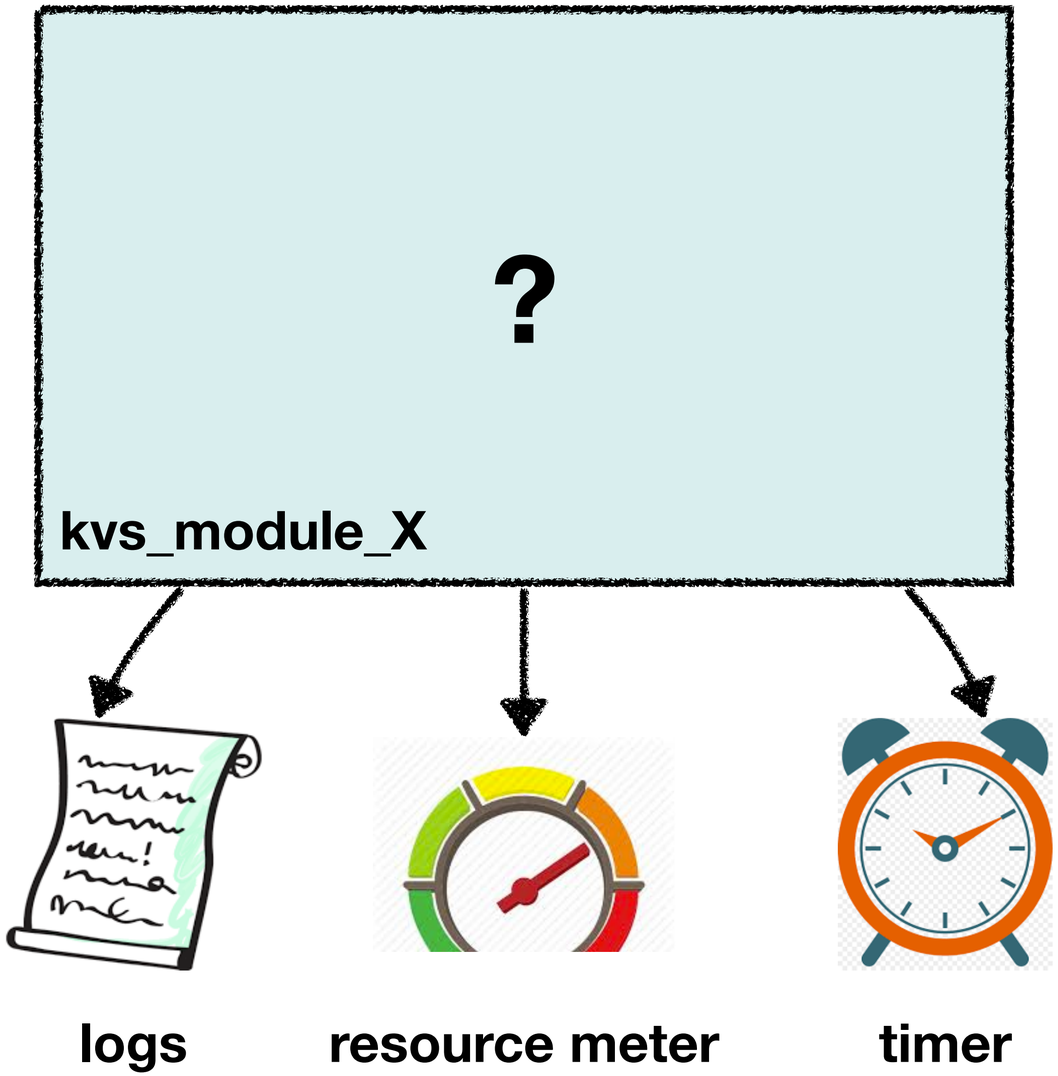
Try #1: Probing



- problem: poor localization
  - e.g. if the response timeouts, we have no idea which step of execution is stuck

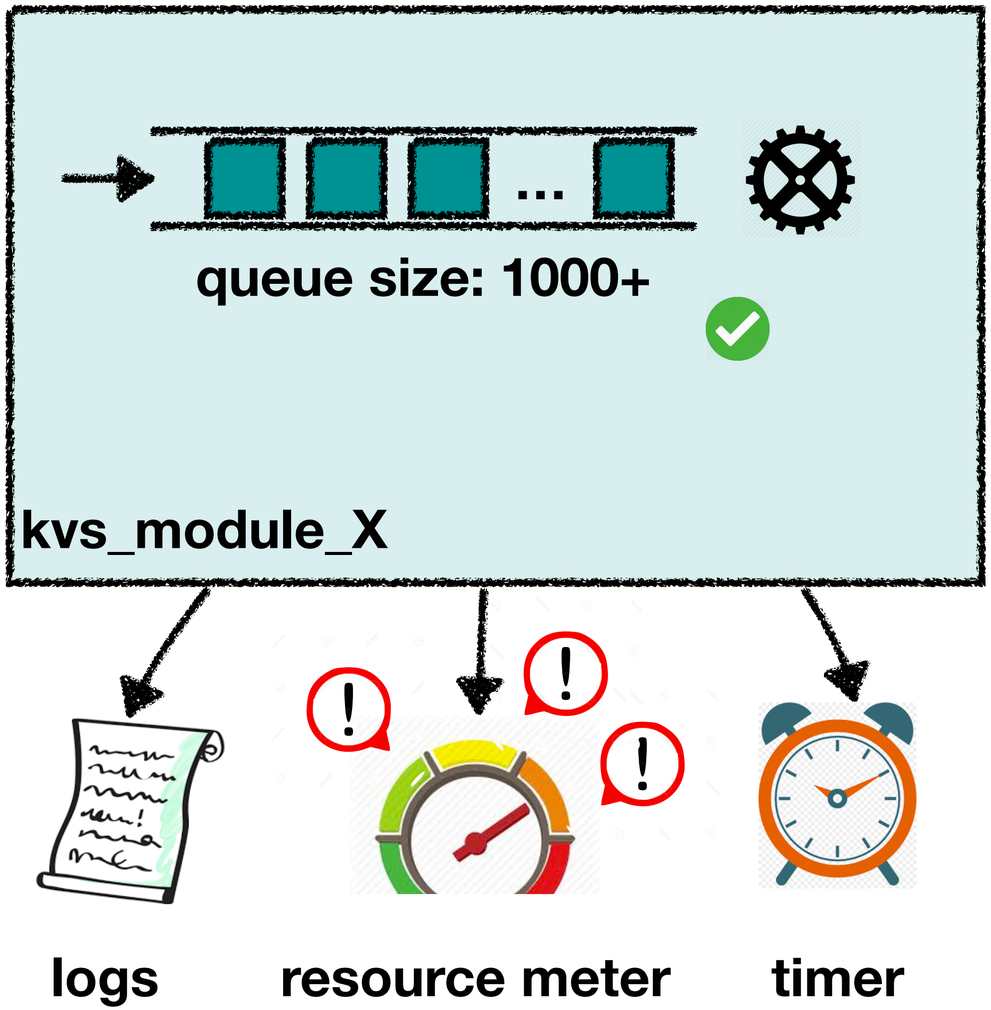
Type	Level	Example	Completeness	Accuracy	Pinpoint
Probe	API	App spy, httpd mod_watchdog	Weak	Perfect	✗

Try #2: Signal



- **define some system health indicators and monitor**
  - e.g. memory load is high ? logs contain ERRORs? process timeout?

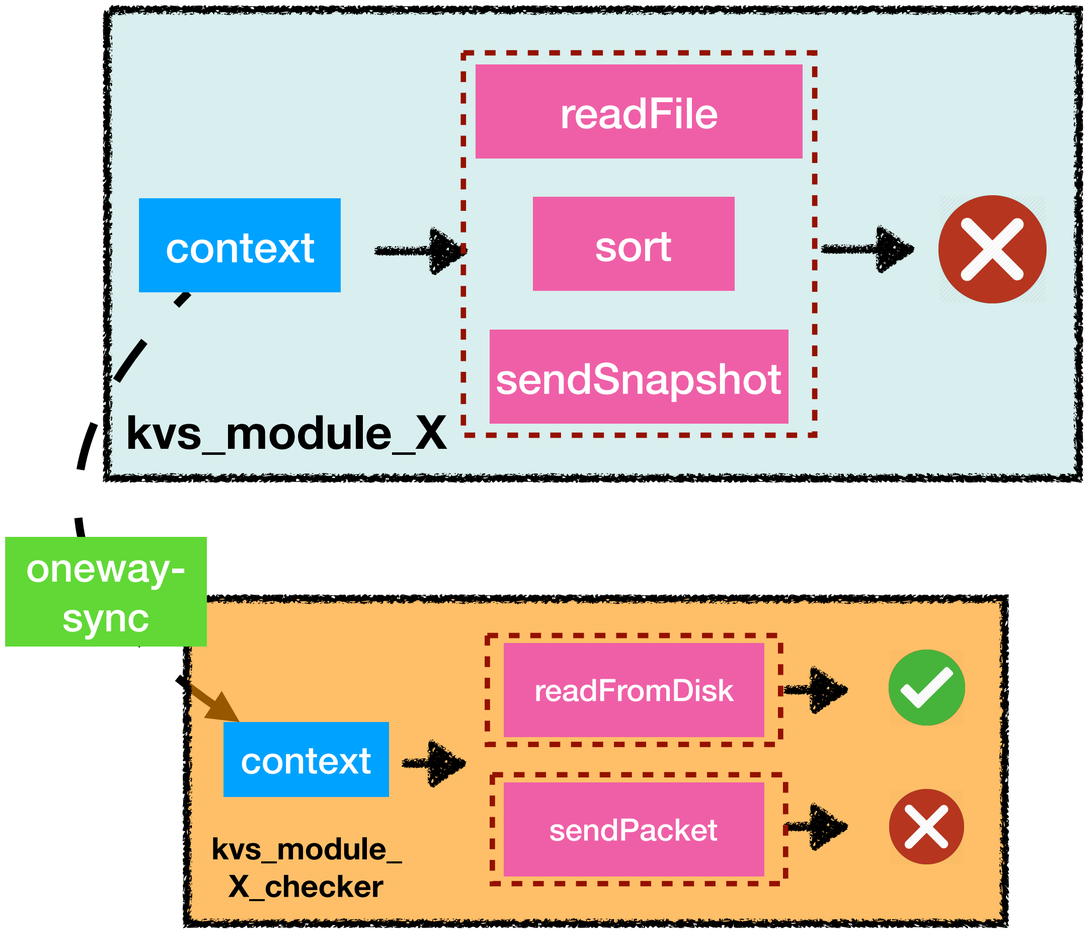
Try #2: Signal



- problem: weak accuracy
  - ✦ excessive signals causing massive false alarms
  - ✦ need significant tuning to be accurate

Type	Level	Example	Completeness	Accuracy	Pinpoint
Probe	API	App spy, httpd mod_watchdog	Weak	Perfect	✗
Signal	Resource	WDT, Linux watchdogd	Modest	Weak	✓

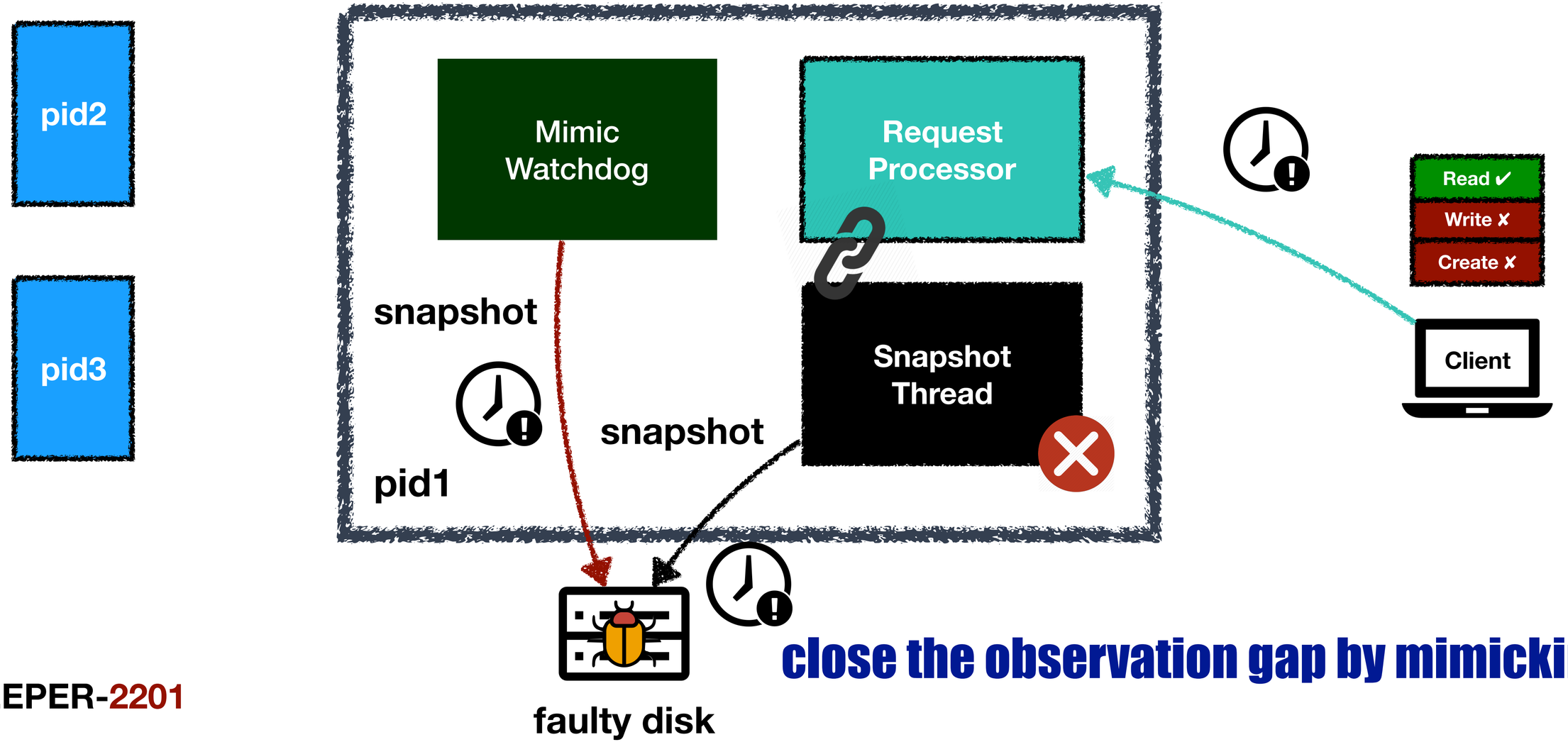
Try #3: Mimic



- imitate what main execution is doing by executing similar operations to expose errors

Type	Level	Example	Completeness	Accuracy	Pinpoint
Probe	API	App spy, httpd mod_watchdog	Weak	Perfect	✗
Signal	Resource	WDT, Linux watchdogd	Modest	Weak	✓
Mimic	Operation	HDFS disk checker (partly)	Strong	Strong	✓

Use mimic checker to detect zookeeper failure



close the observation gap by mimicking!

ZOOKEEPER-2201

Challenges to write mimic-type watchdogs

- **time-consuming for developers to manually write good watchdogs**
  - too many modules and functions to be covered
- **challenging to write it right**
  - e.g. alter the main execution, invoke a dangerous operation

Outline

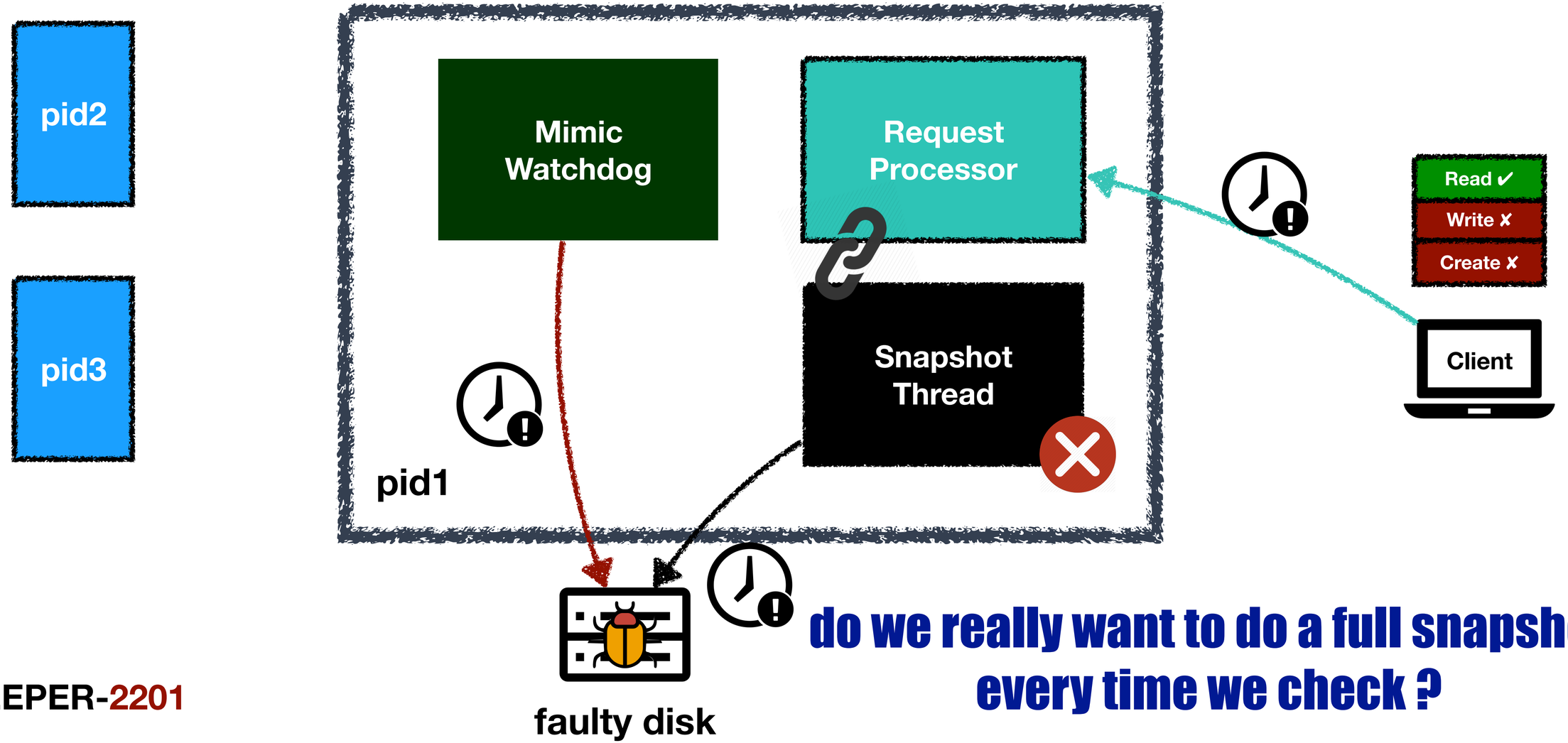
- Motivation
- Intrinsic software watchdog abstraction
  - hardware & software watchdogs
  - characteristics
  - checker approach
- AutoWatchdog: a tool to generate watchdogs
  - technique: program reduction
- Challenges & Opportunities

Our system

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

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

Why do program reduction?



ZOOKEEPER-2201

do we really want to do a full snapshot every time we check ?

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);
        }
    }
}
```

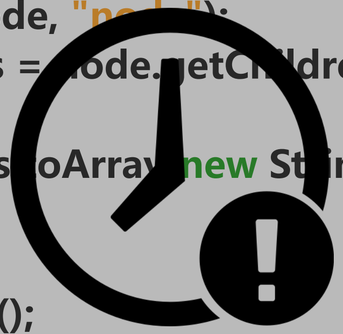
Mimic Watchdog

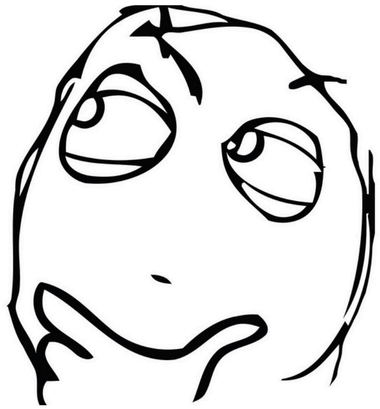
what if put the whole snapshot operation into the checker and run?

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

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

```
@Override
public void run() {
    try {
        int logCount = 0;

        // we do this in an attempt to ensure that not all of the servers
        // in the ensemble take a snapshot at the same time
        setRandRoll(r.nextInt( bound: snapCount/2));
        while (true) {
            Request si = null;
            if (toFlush.isEmpty()) {
                si = queuedRequests.take();
            } else {
                si = queuedRequests.poll();
                if (si == null) {
                    flush(toFlush);
                    continue;
                }
            }
            if (si == requestOfDeath) {
                break;
            }
            if (si != null) {
                // track the number of records written to the log
                if (zks.getZKDatabase().append(si)) {
                    logCount++;
                    if (logCount > (snapCount / 2 + randRoll)) {
                        randRoll = r.nextInt( bound: snapCount/2);
                        // roll the log
                        zks.getZKDatabase().rollLog();
                        // take a snapshot
                        if (snapInProcess != null && snapInProcess.isAlive()) {
                            LOG.warn("Too busy to snap, skipping");
                        } else {
                            new Error().printStackTrace();
                            snapInProcess = new Thread( name: "Snapshot Thread") {
                                    public void run() {
                                        try {
                                            zks.takeSnapshot();
                                        } catch(Exception e) {
                                            LOG.warn("Unexpected exception", e);
                                        }
                                    }
                                };
                            snapInProcess.start();
                        }
                        logCount = 0;
                    }
                } else if (toFlush.isEmpty()) {
                    // optimization for read heavy workloads
                    // iff this is a read, and there are no pending
                    // flushes (writes), then just pass this to the next
                    // processor
                    if (nextProcessor != null) {
                        nextProcessor.processRequest(si);
                        if (nextProcessor instanceof Flushable) {
                            ((Flushable)nextProcessor).flush();
                        }
                    }
                    continue;
                }
                toFlush.add(si);
                if (toFlush.size() > 1000) {
                    flush(toFlush);
                }
            }
        }
    } catch (Throwable t) {
        LOG.error("Severe unrecoverable error, exiting", t);
        running = false;
        System.exit( status: 11);
    }
    LOG.info("SyncRequestProcessor exited!");
}
```

Step#1 locate long-running regions

```
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!");
  }
```

initialization stage

long-running stage

cleanup stage

Step#2 reduce the program

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

keep reducing

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

Preliminary results

- **AutoWatchdog**:
  - **based on Soot and supports Java programs**
  - **applied to Zookeeper, HDFS, Cassandra**
  - **successfully detected ZooKeeper-2201 failure in ~7 seconds**
    - with blocked function pinpointed and concrete context captured
  - **with moderate performance overhead of 7.2 % averagely**
    - compared to 1.5% for the probing checker

Outline

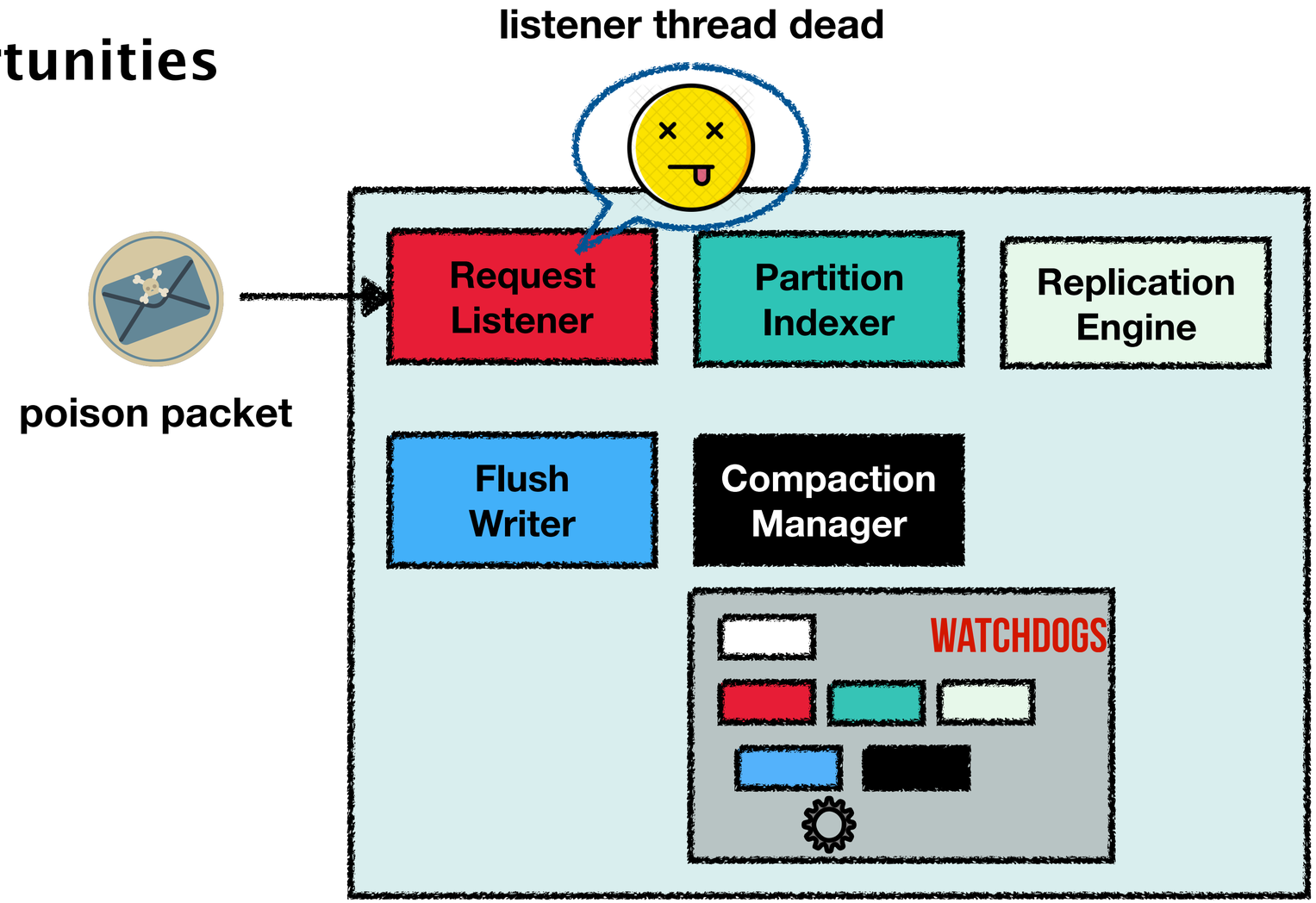
- **Motivation**
- **Intrinsic software watchdog abstraction**
  - hardware & software watchdogs
  - characteristics
  - checker approach
- **AutoWatchdog: a tool to generate watchdogs**
  - technique: program reduction
- **Challenges & Opportunities**

Discussion

- **Challenges**
  - **locating vulnerable operations is heuristic based**
    - a more principled algorithm to select vulnerable operations?
  - **assess the impact of the detected fault**
    - invoke a validator (probing?) upon failure detection?
  - **semantic checks (fsck-like checks)**
    - leverage test cases to mine semantic checks?

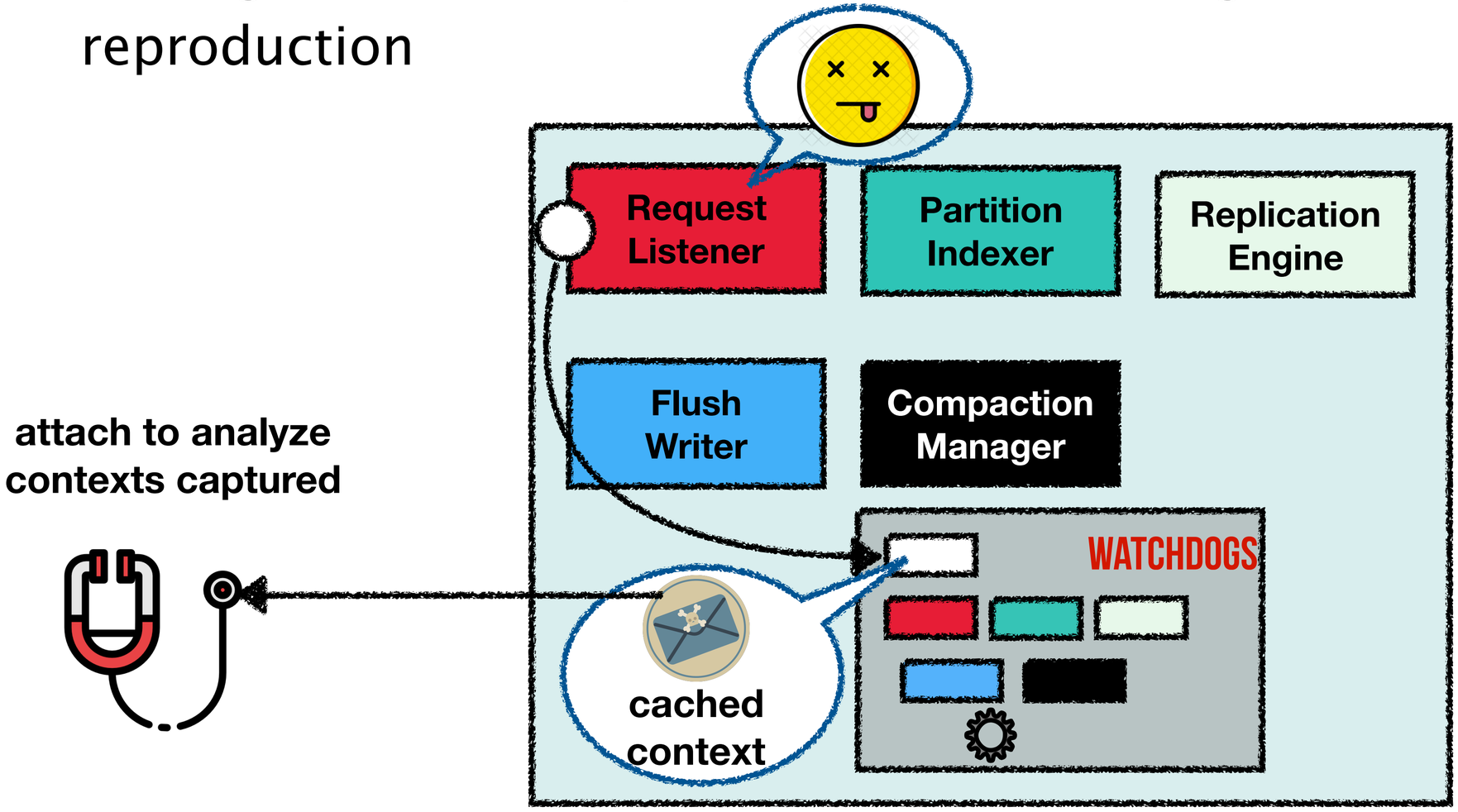
Discussion

- Opportunities



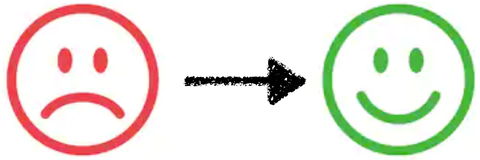
Discussion

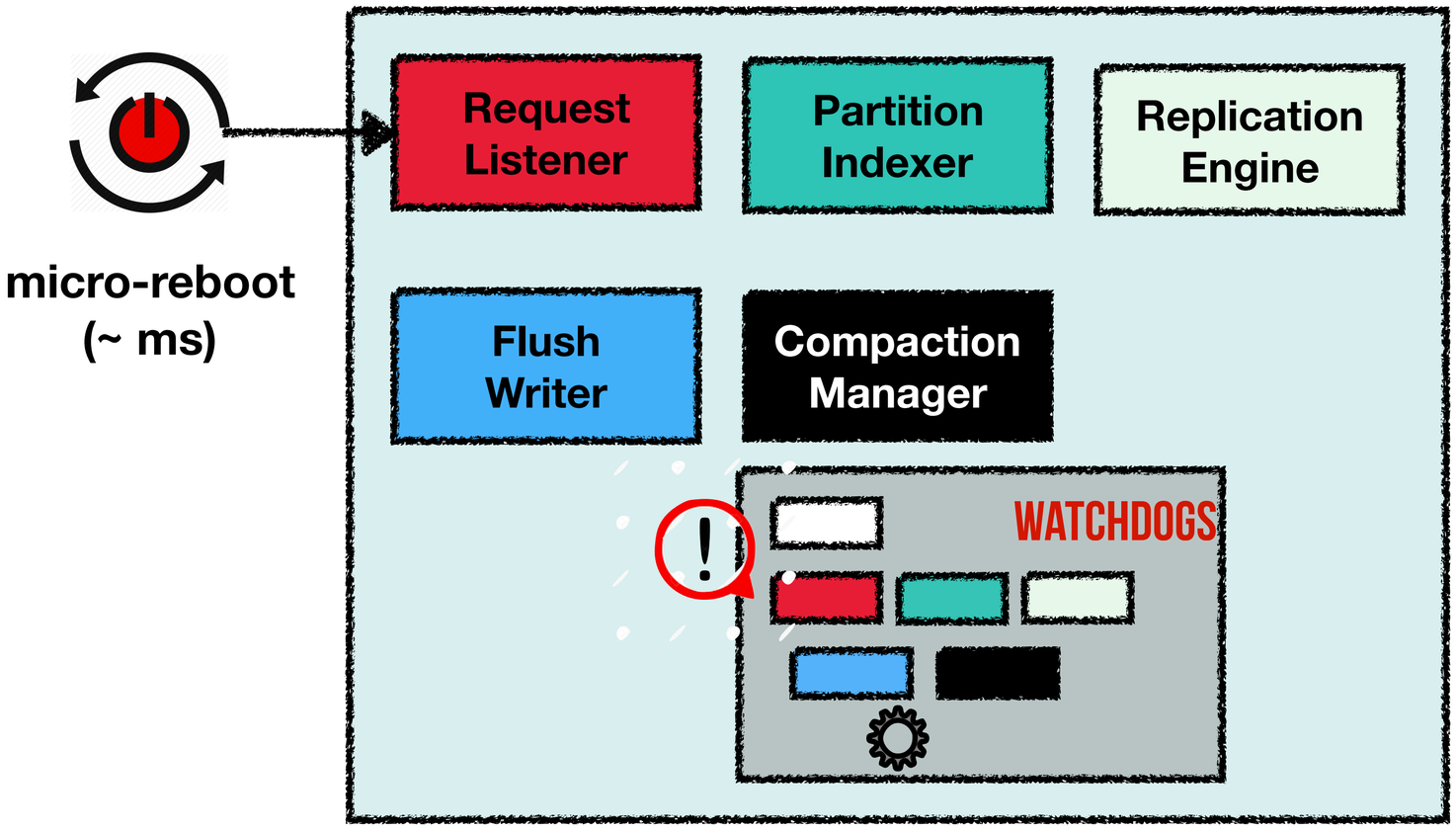
- **Opportunities**
  - leverage contexts captured by the watchdogs for failure reproduction



Discussion

- **Opportunities**
  - cheap recovery





Conclusion

- **Modern software are increasingly complex and often fail partially**
  - these subtle failures cannot be detected by process-level failure detectors
- **We propose an intrinsic software watchdog abstraction**
  - three characteristics: tailored, stateful and concurrent checkers
- **Mimic-type checkers expose failures by imitating main program**
  - good accuracy, completeness and localization, but challenging to write manually
- **AutoWatchdog generates intrinsic watchdogs with mimic checkers**
  - core technique: program reduction