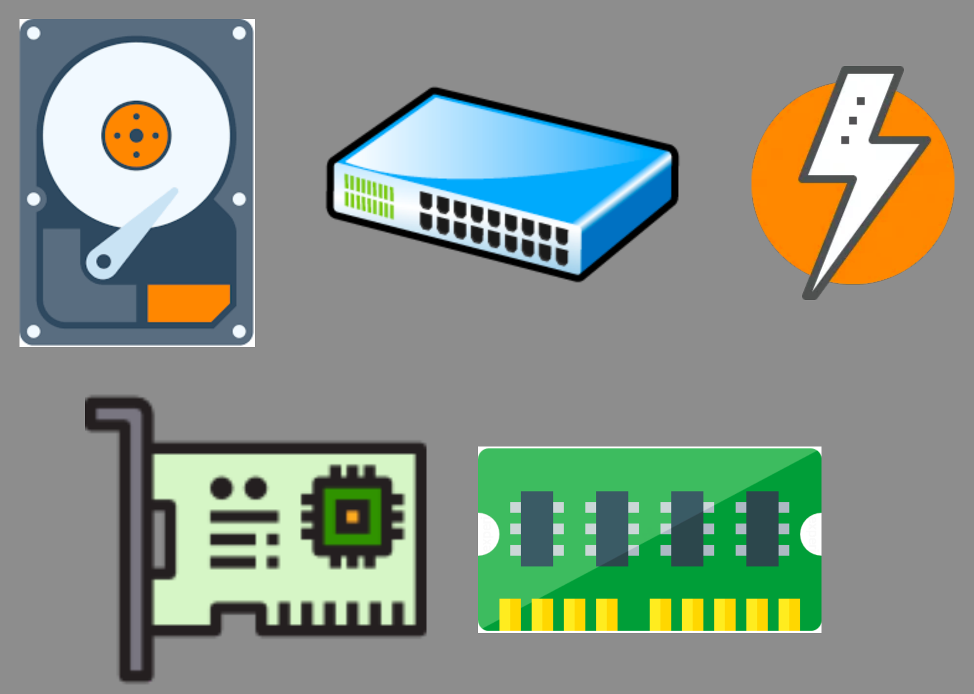
Capturing and Enhancing *In Situ* System Observability for Failure Detection

Ryan Huang*

Chuanxiong Guo†, Jacob R. Lorch‡, Lidong Zhou‡, Yingnong Dang+

*Johns Hopkins University †ByteDance Inc. ‡ Microsoft Research + Microsoft

Faults Are Common in Large Systems



Hardware Faults

```
int len = in.readInt();
if (len == -1)
  return null;
byte arr[] = new byte[len];
in.readFully(arr);
return arr;
```

Software Bugs

Misconfiguration

Detecting Failure Is Crucial

FD is a fundamental building block for fault-tolerant systems

$$Availability \uparrow \approx \frac{MTBF \uparrow}{MTBF + MTTR \downarrow}$$

Improve reliability

Speed up recovery[1]

[1] Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies

Detecting Failure Is Crucial

FD is a fundamental building block for fault-tolerant systems

$$Availability \uparrow \approx \frac{MTBF \uparrow}{MTBF + MTTR \downarrow}$$

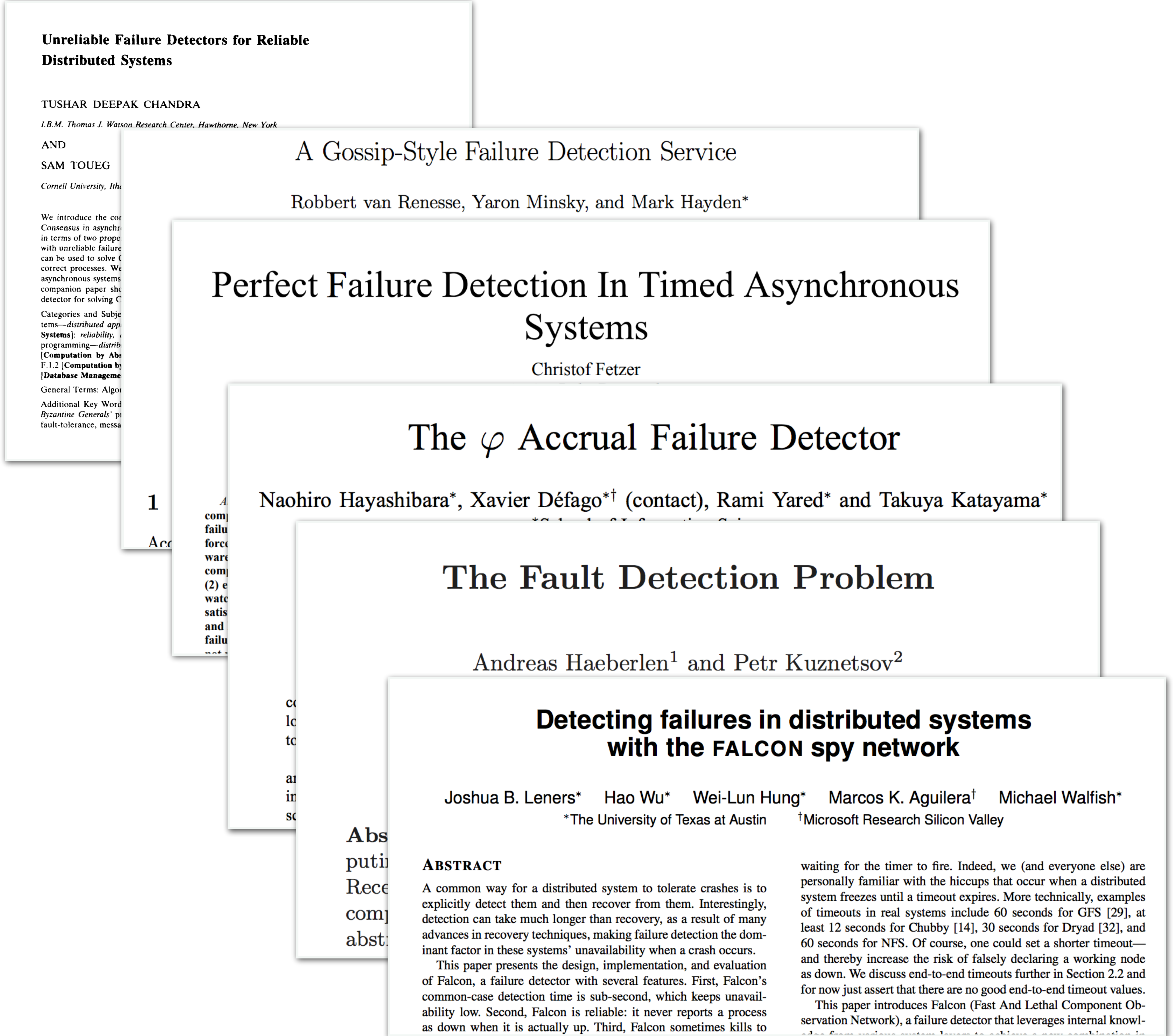
Improve reliability

Speed up recovery[1]

But only if failure can be
detected reliably and rapidly first

[1] Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies

Failure Detection is Hard

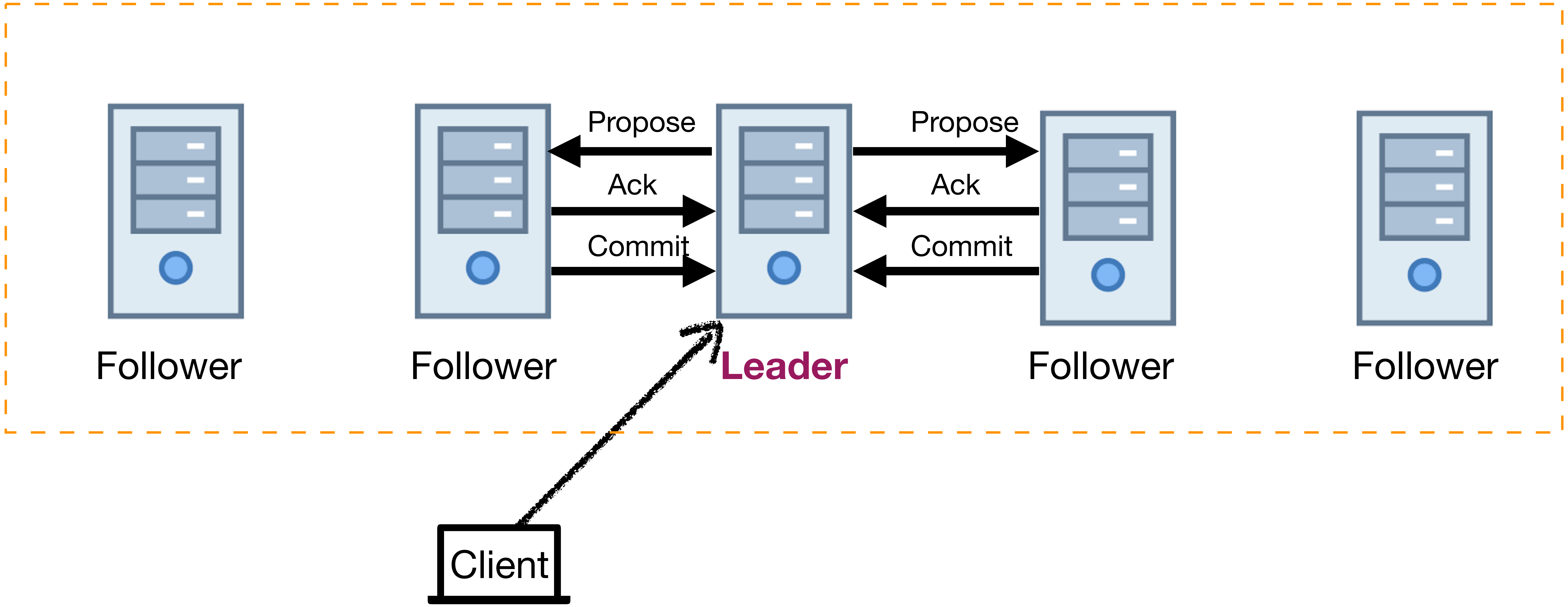


- Extensively studied for several decades
- Focus on detection of fail-stop failures in asynchronous systems
  - Difficult to reliably determine whether a process has crashed or is merely "very slow"

Production Failures Beyond Fail-Stop

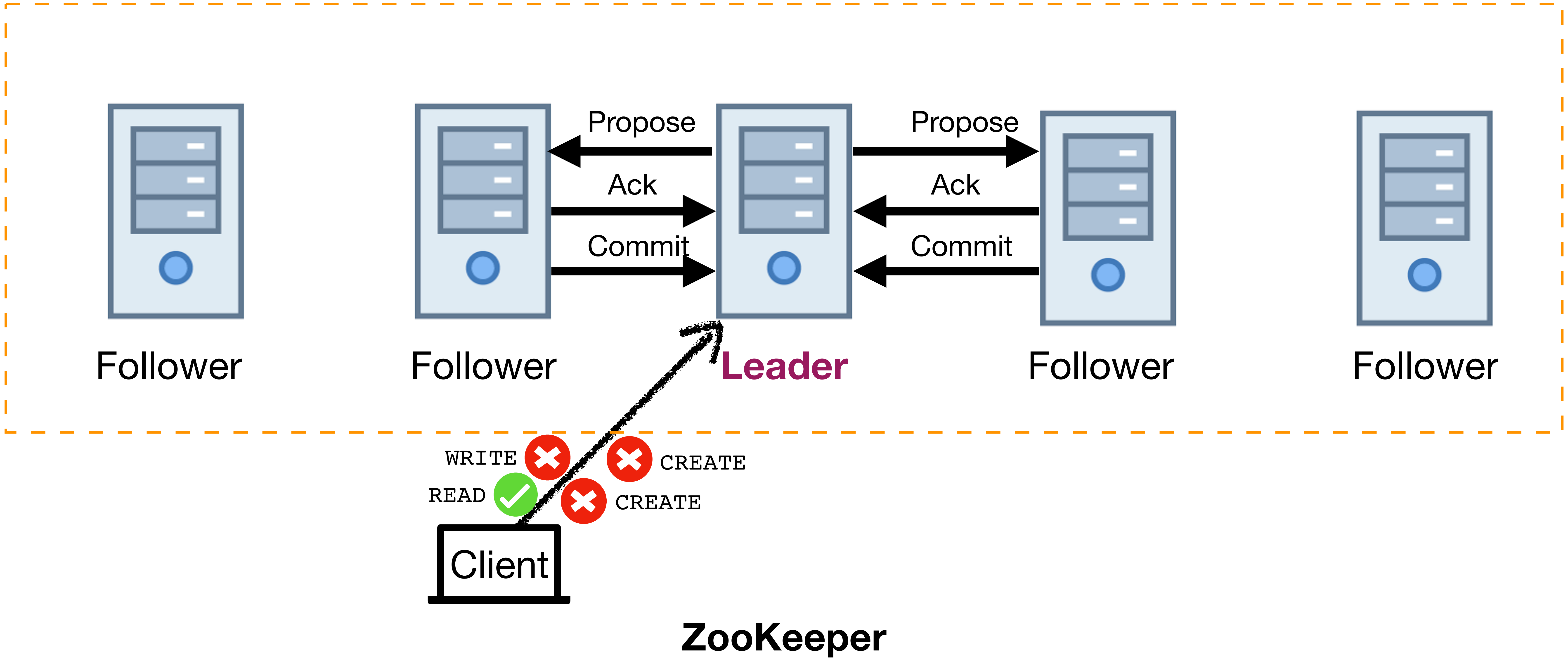
- A component status can be between `UP` and `DOWN`: gray failure
  - E.g., fail-slow hardware, random packet loss, limp lock, ...
  - Failure symptoms are very subtle
- Common in production cloud systems
- Pose new challenges for failure detection

An Example Production Gray Failure

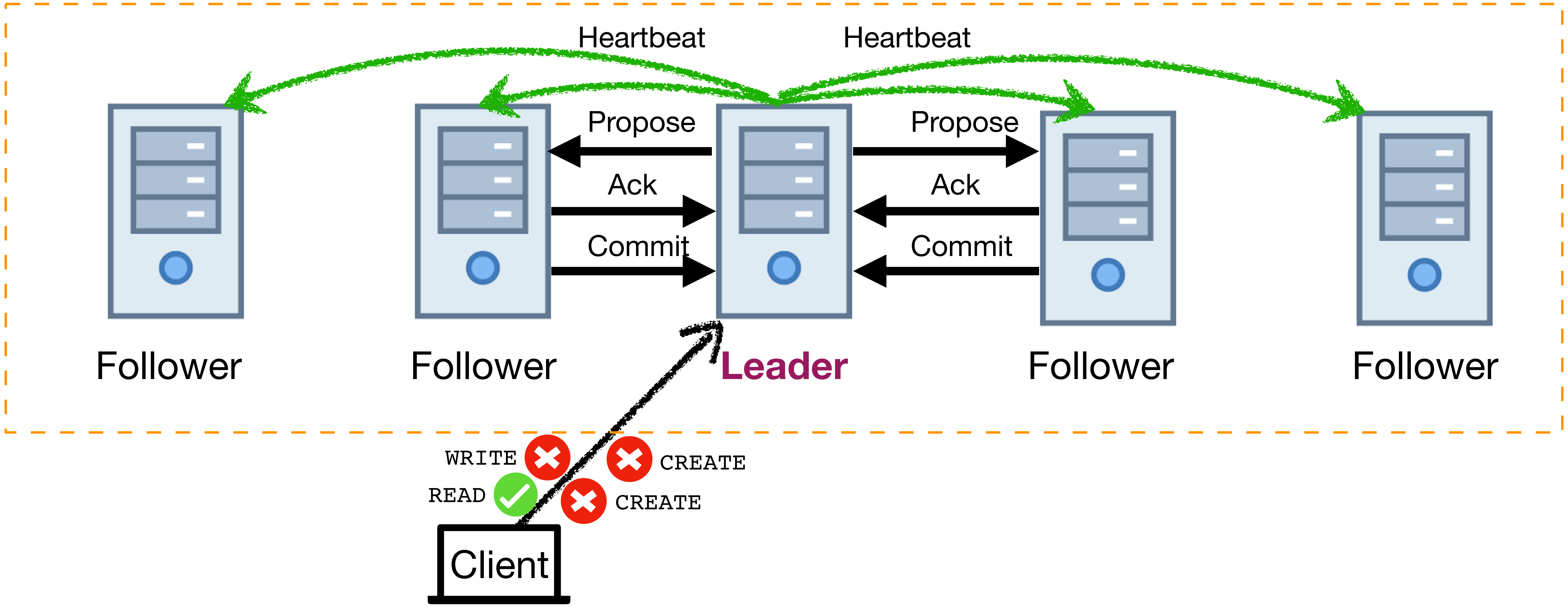


ZooKeeper

An Example Production Gray Failure

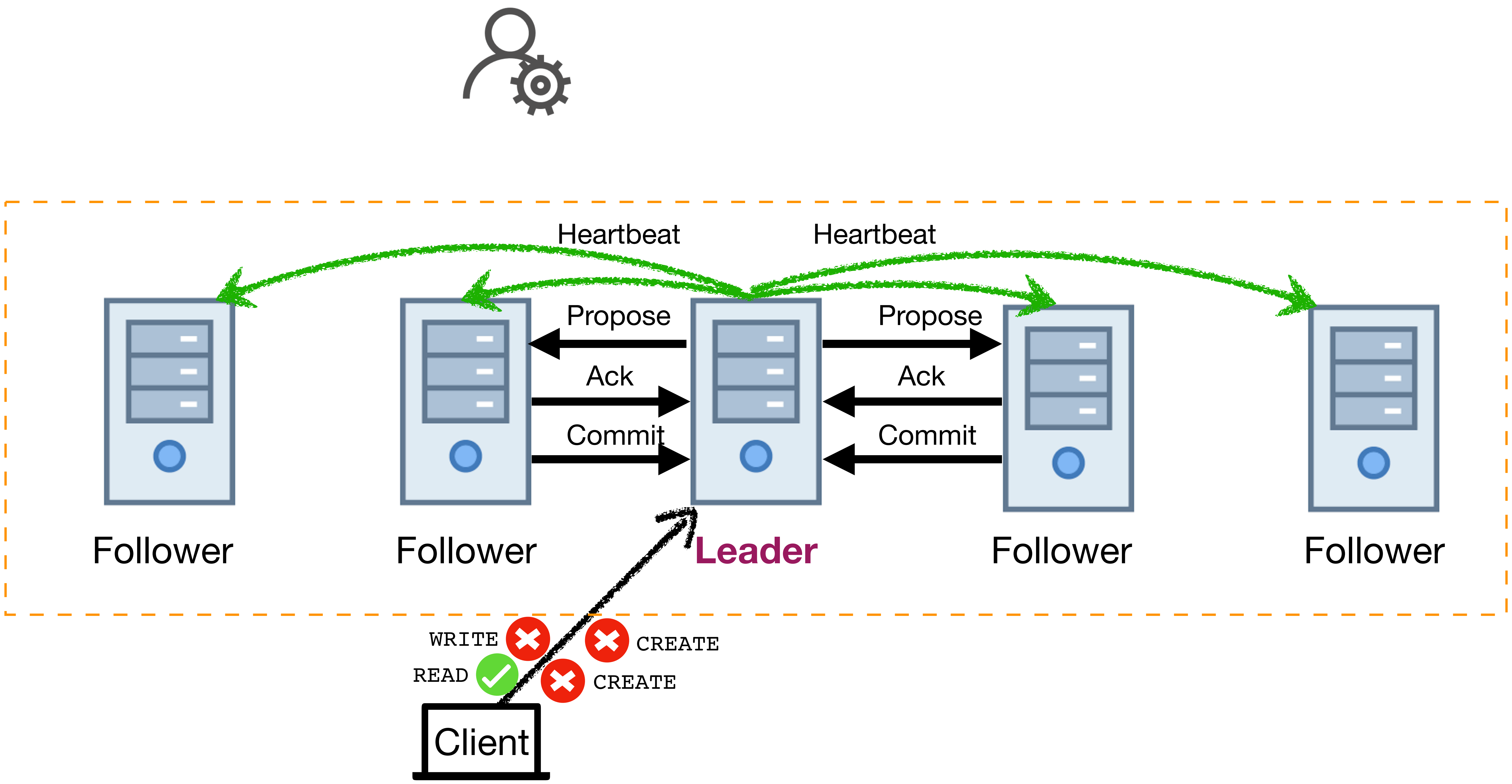


An Example Production Gray Failure



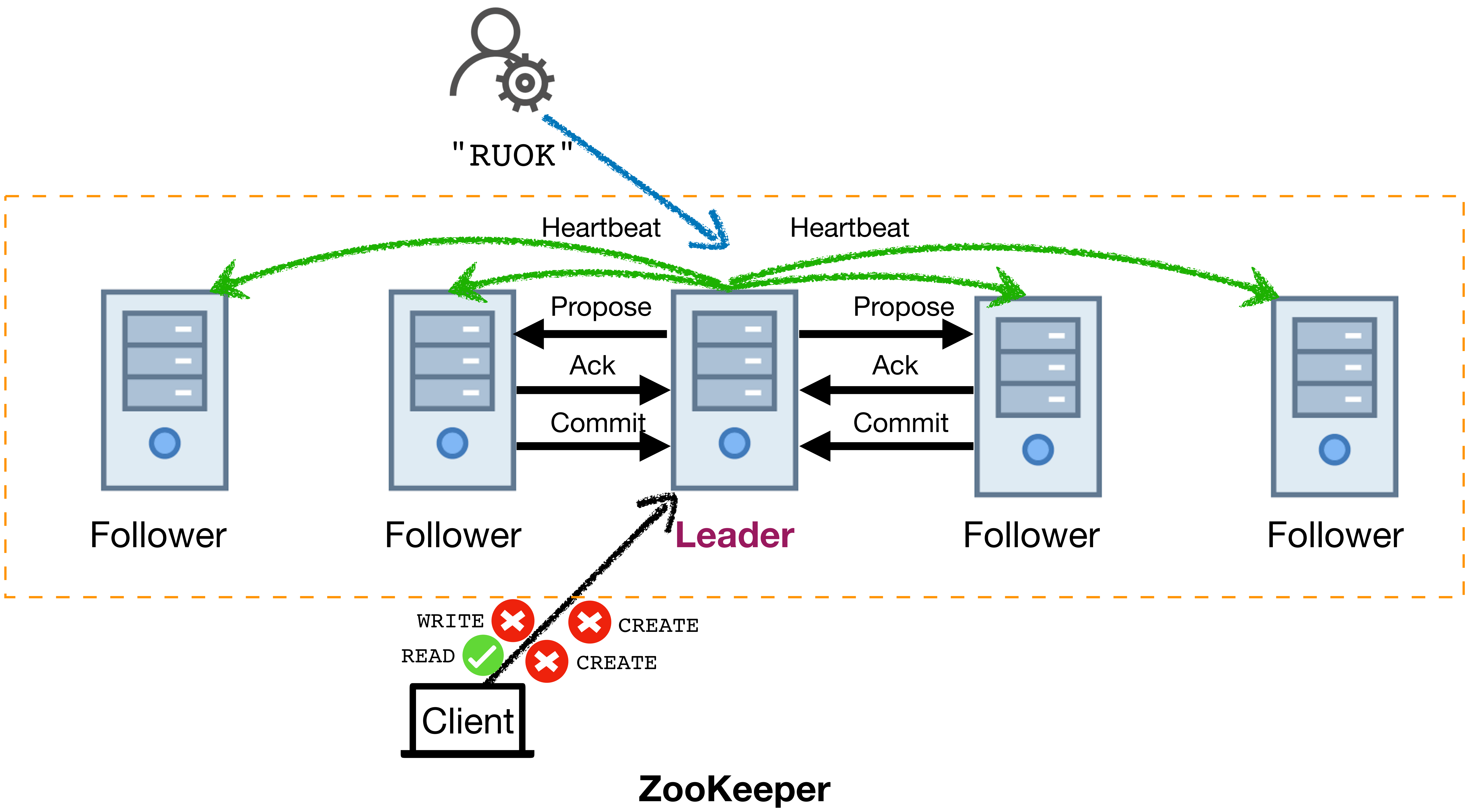
ZooKeeper

An Example Production Gray Failure

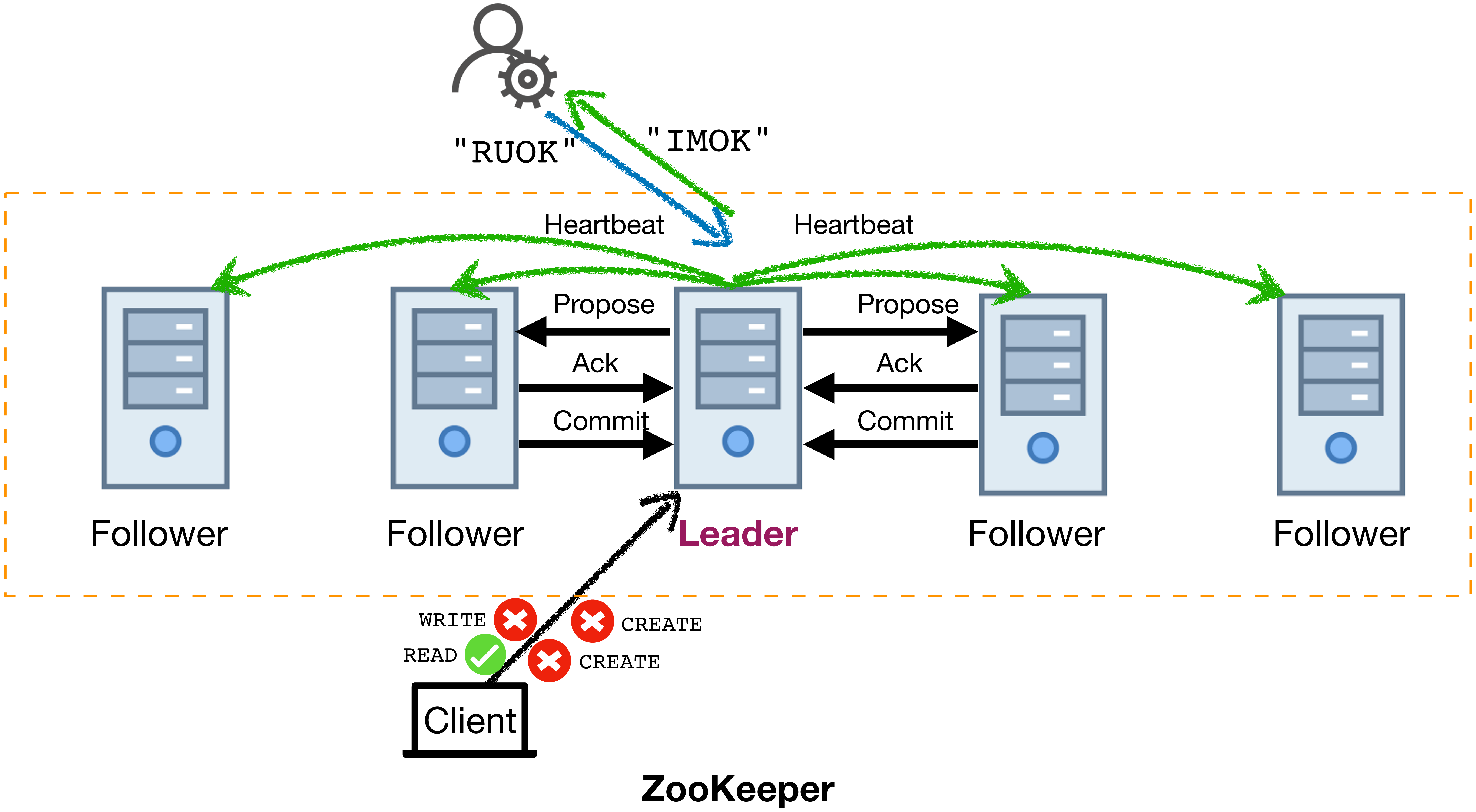


ZooKeeper

An Example Production Gray Failure



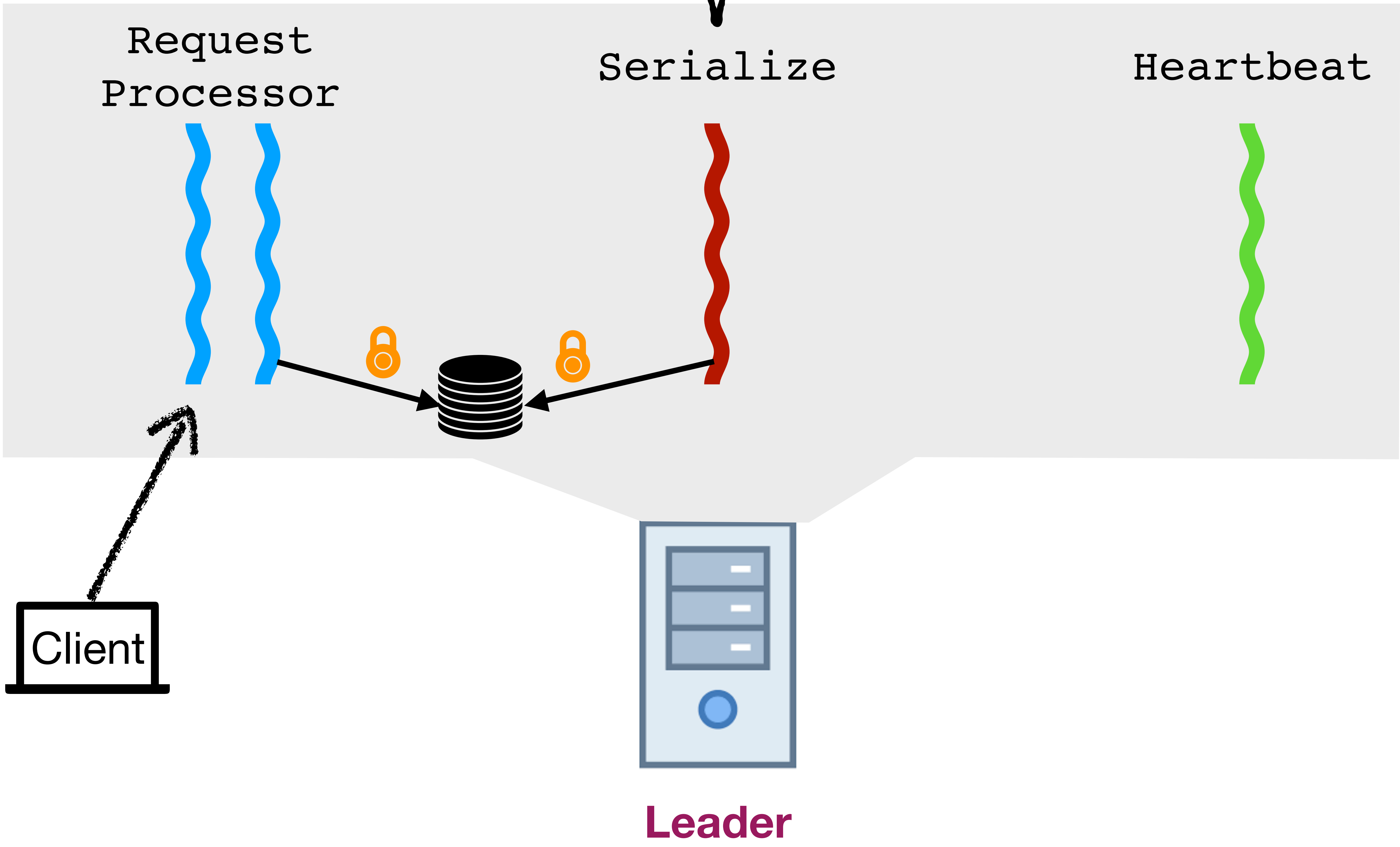
An Example Production Gray Failure

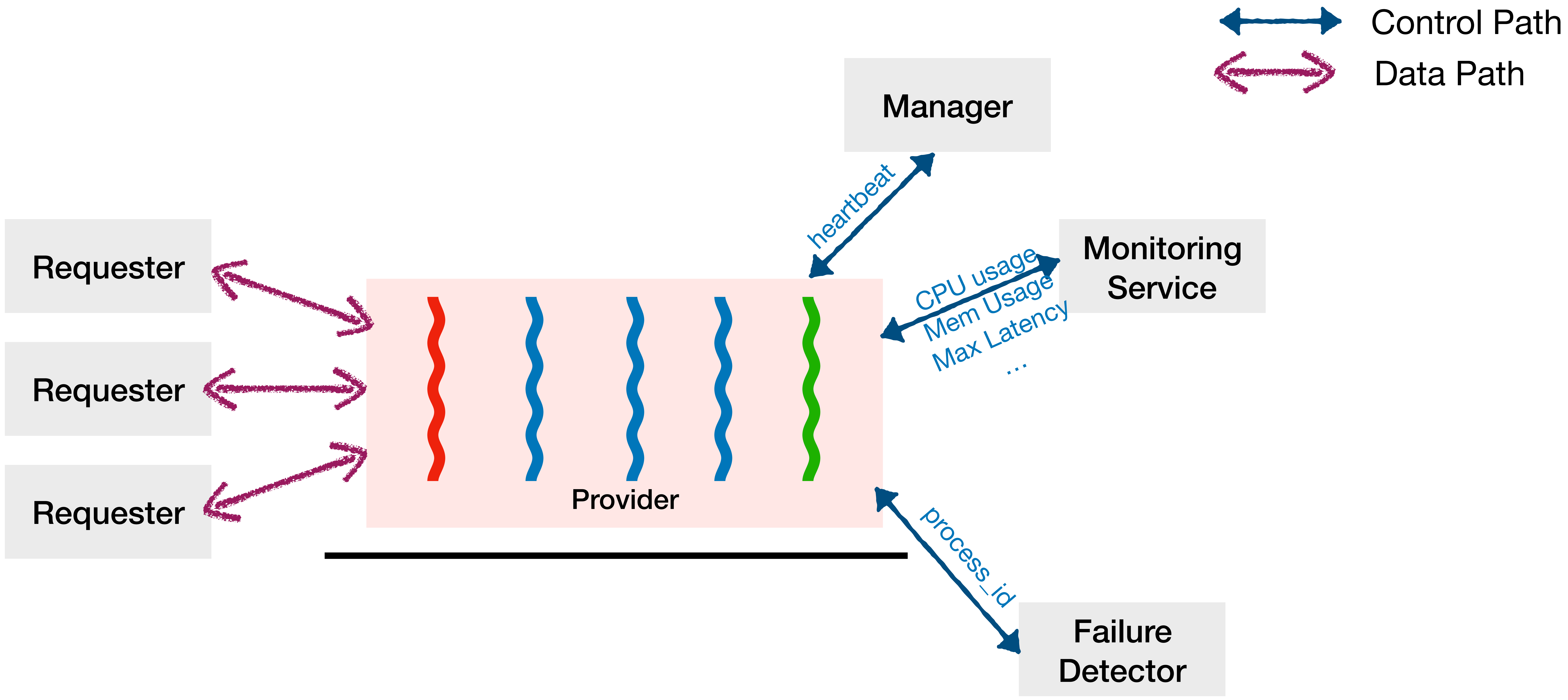


Root Cause of the Gray Failure

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

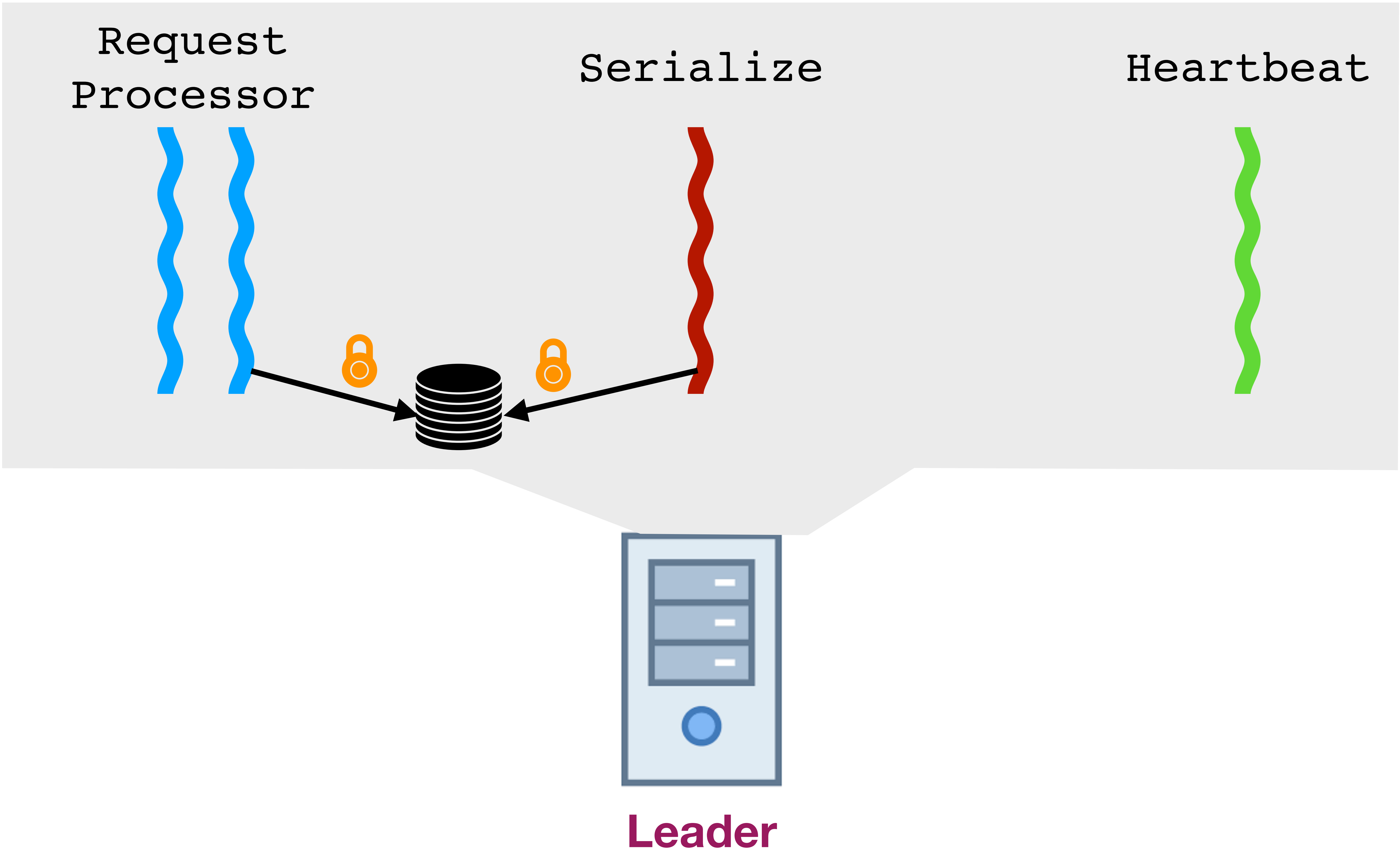
Stuck due to transient network issue



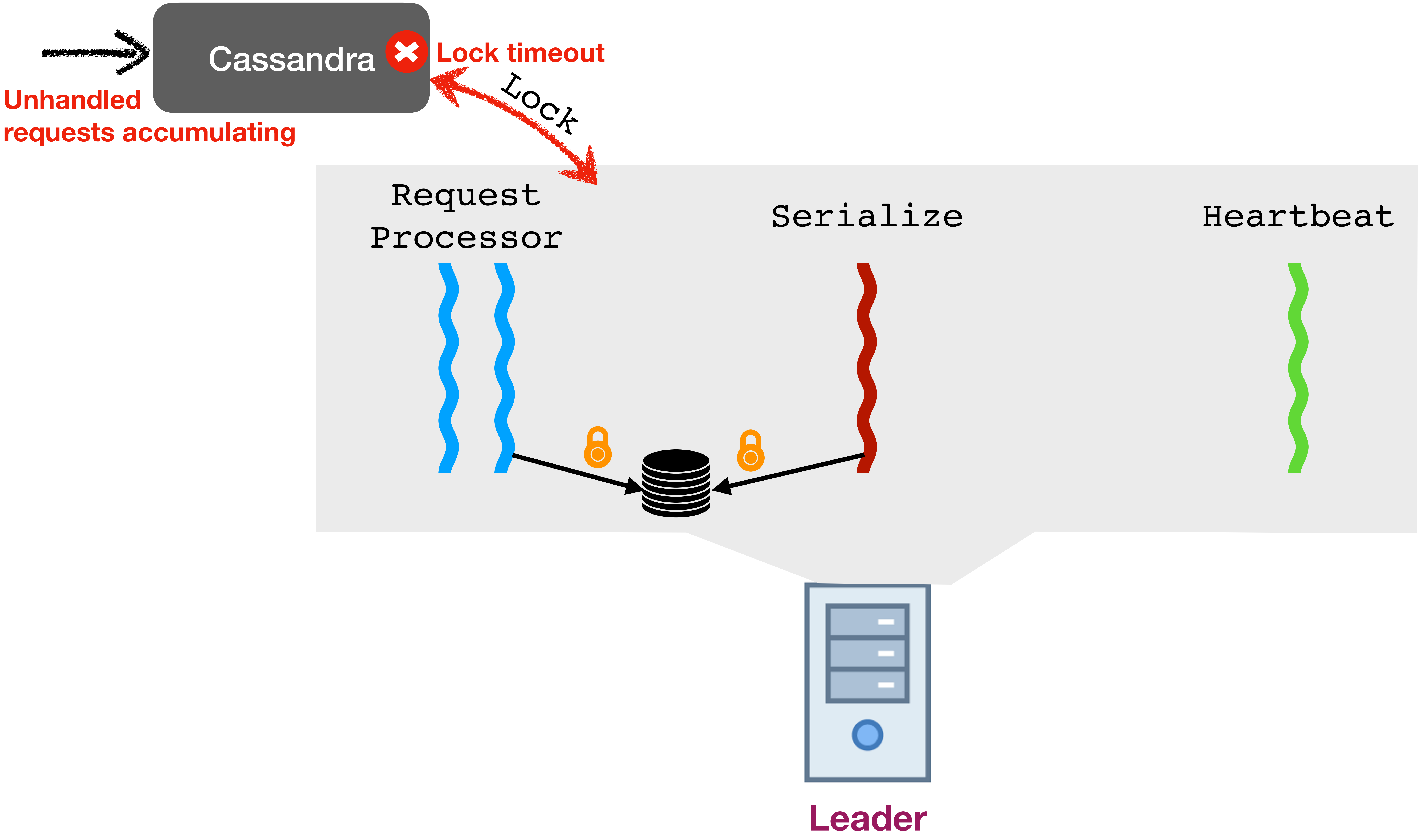


Insight: we should detect what the *requesters* see

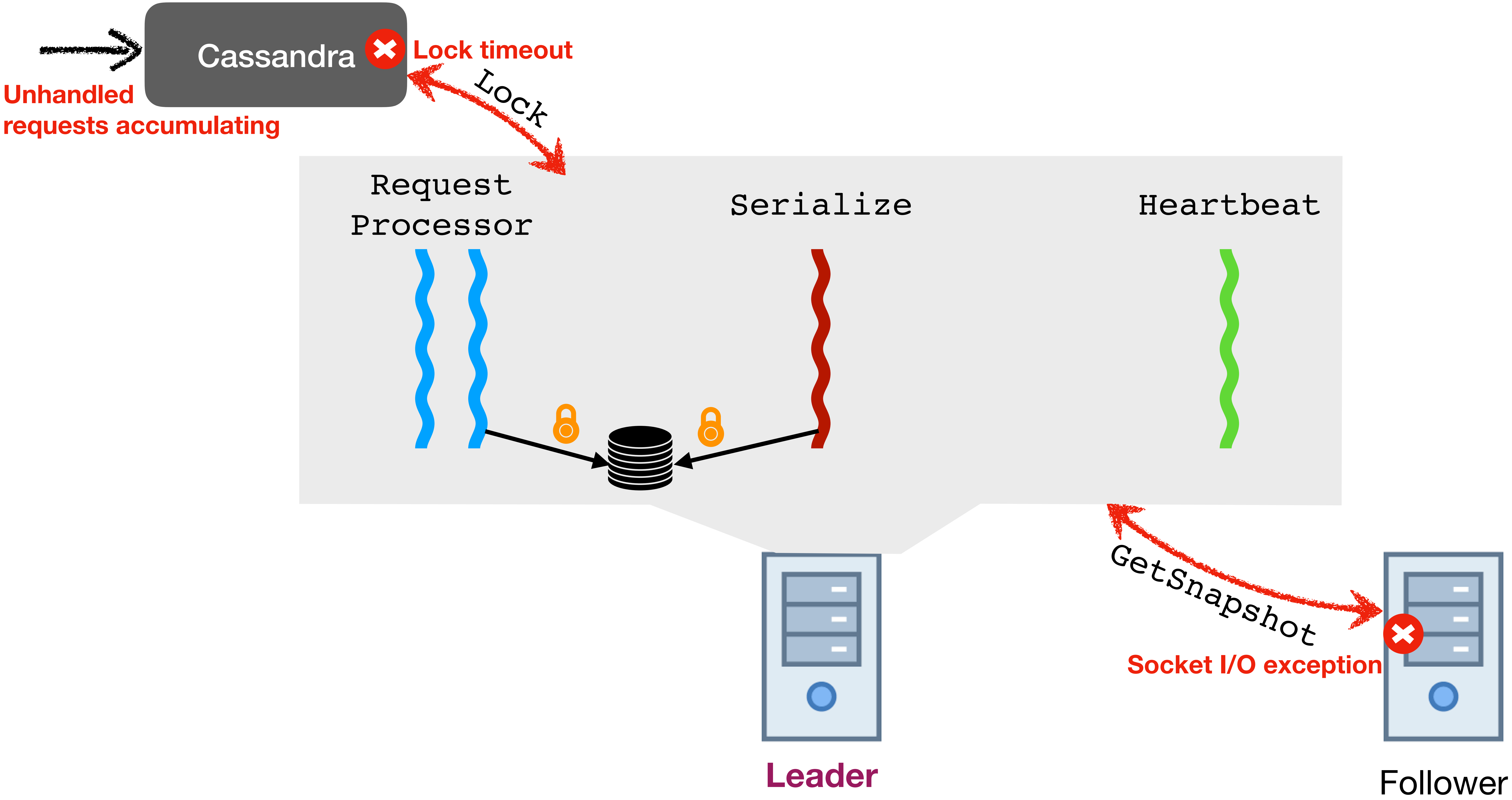
Insight 1: Critical Gray Failures Are *Observable*



Insight 1: Critical Gray Failures Are *Observable*



Insight 1: Critical Gray Failures Are *Observable*



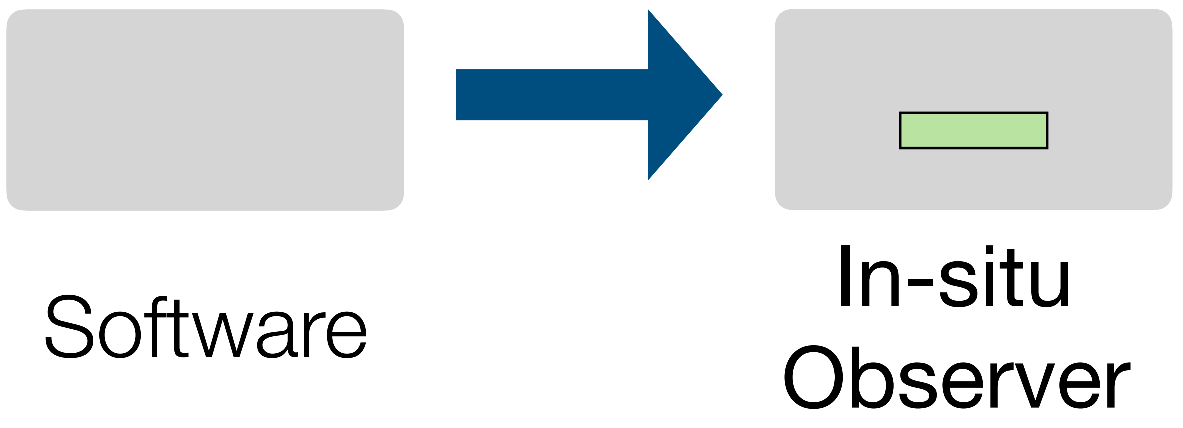
Insight 2: From Error *Handling* to Error *Reporting*

```
void syncWithLeader(long newLeaderZxid) {
   try {
        deserializeSnapshot(leaderIs);
        String sig = leaderIs.read("signature");
        if (!sig.equals("BenWasHere"))
            throw new IOException("Bad signature");
        } else {
            LOG.error("Unexpected leader packet.");
            System.exit(13);
        }
    catch (IOException e) {
        LOG.warn("Exception sync with leader", e);
        sock.close();
    }
}
```

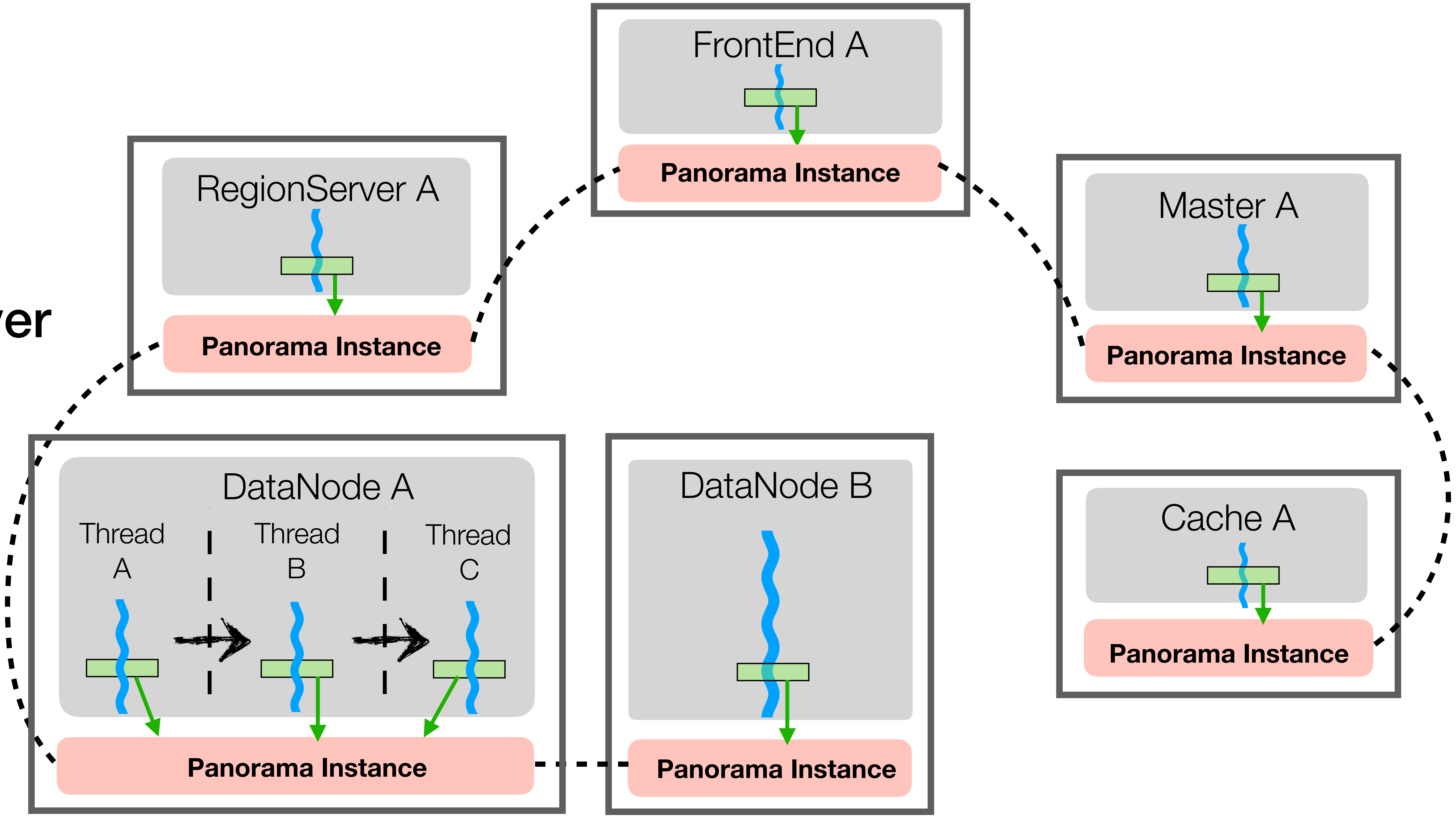
Insight 2: From Error *Handling* to Error *Reporting*

```
void syncWithLeader(long newLeaderZxid) {
   try {
        deserializeSnapshot(leaderIs);
        String sig = leaderIs.read("signature");
        if (!sig.equals("BenWasHere"))
            throw new IOException("Bad signature");
        } else {
            LOG.error("Unexpected leader packet.");
            System.exit(13);
        }
    catch (IOException e) {
        LOG.warn("Exception sync with leader", e);
        sock.close();
    }
}
```

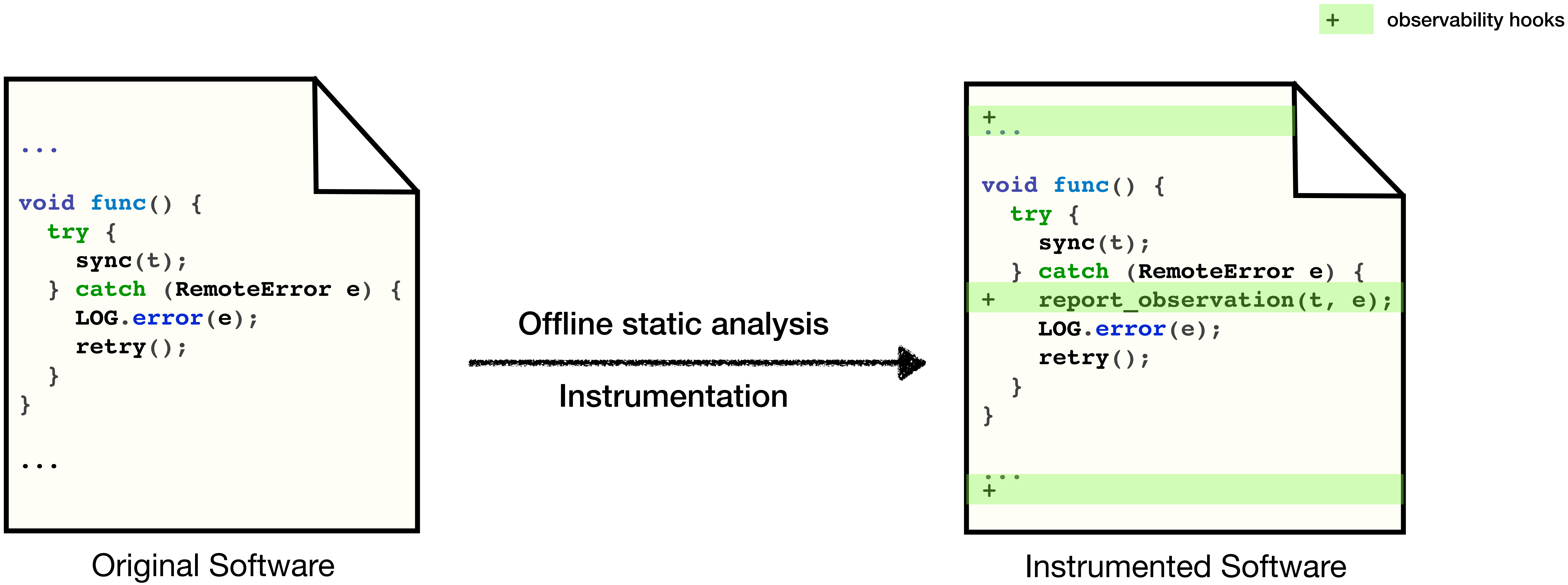
Our Solution: Panorama



- A tool to turn a software into an observer
- Uniform observation abstractions
- A generic failure detection service for any component to participate

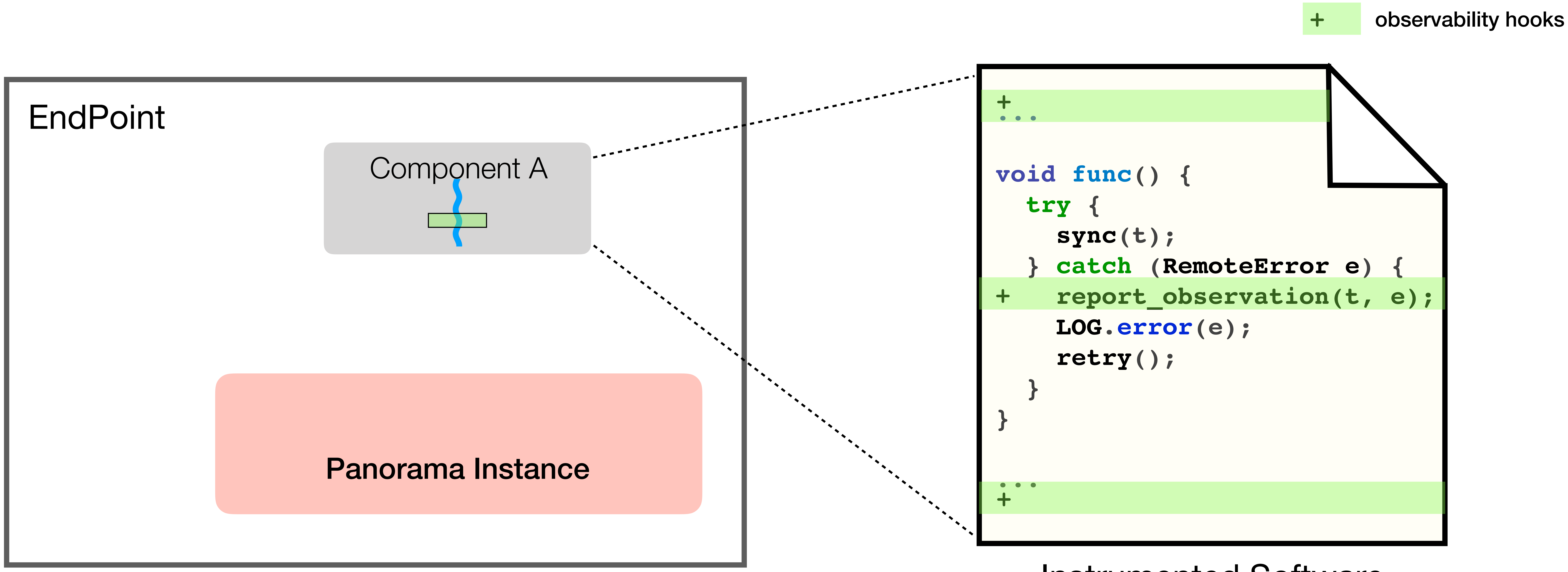


Overview of Panorama - Analysis



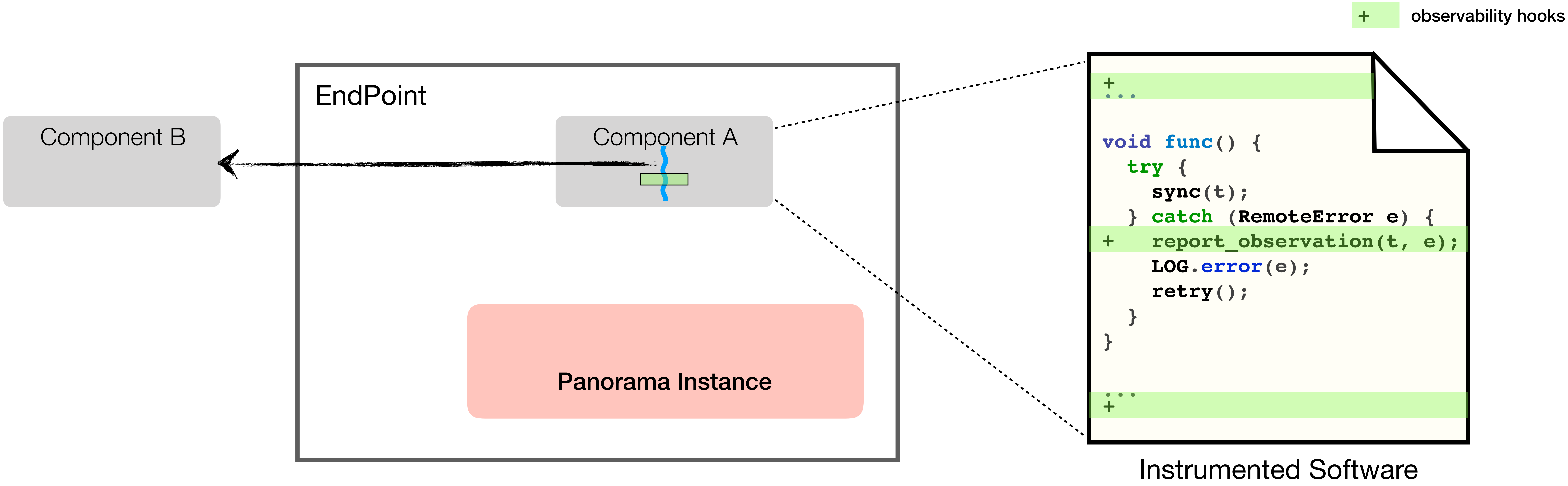
Automatically convert a software component into an *in-situ* observer

Overview of Panorama - Runtime

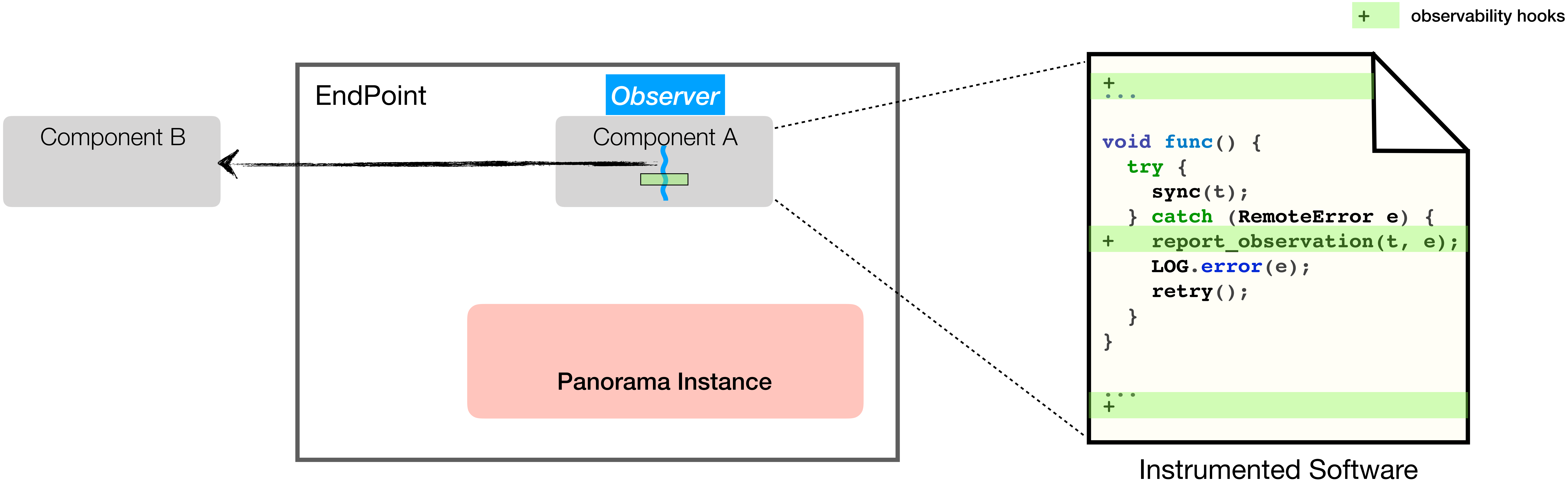


Instrumented Software

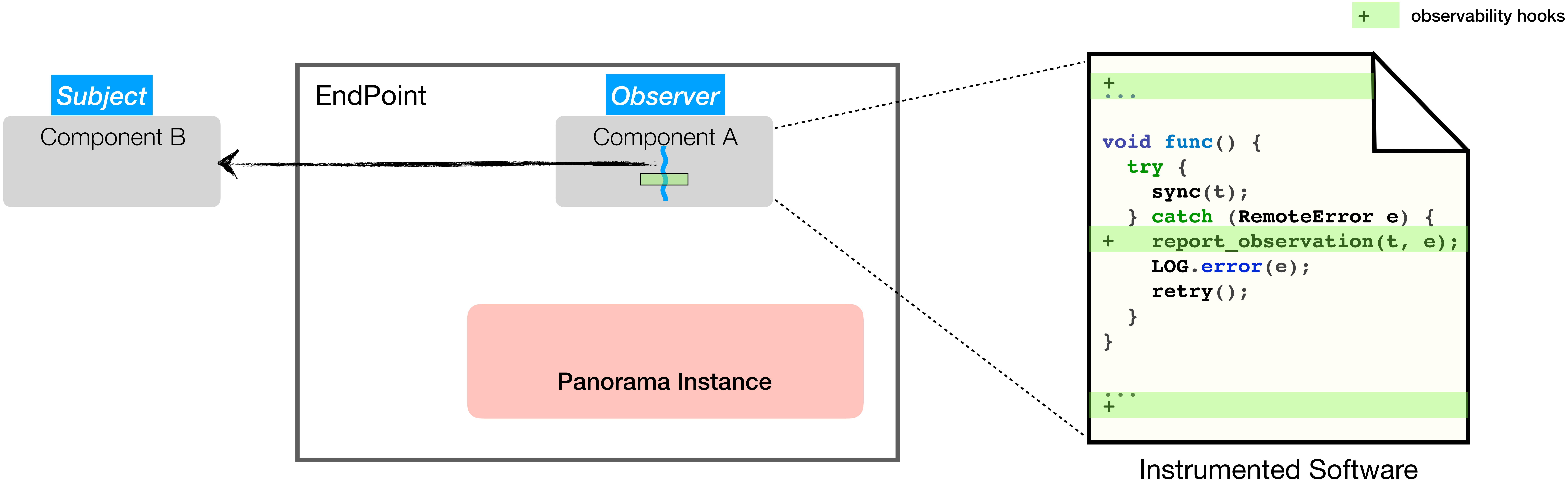
Overview of Panorama - Runtime



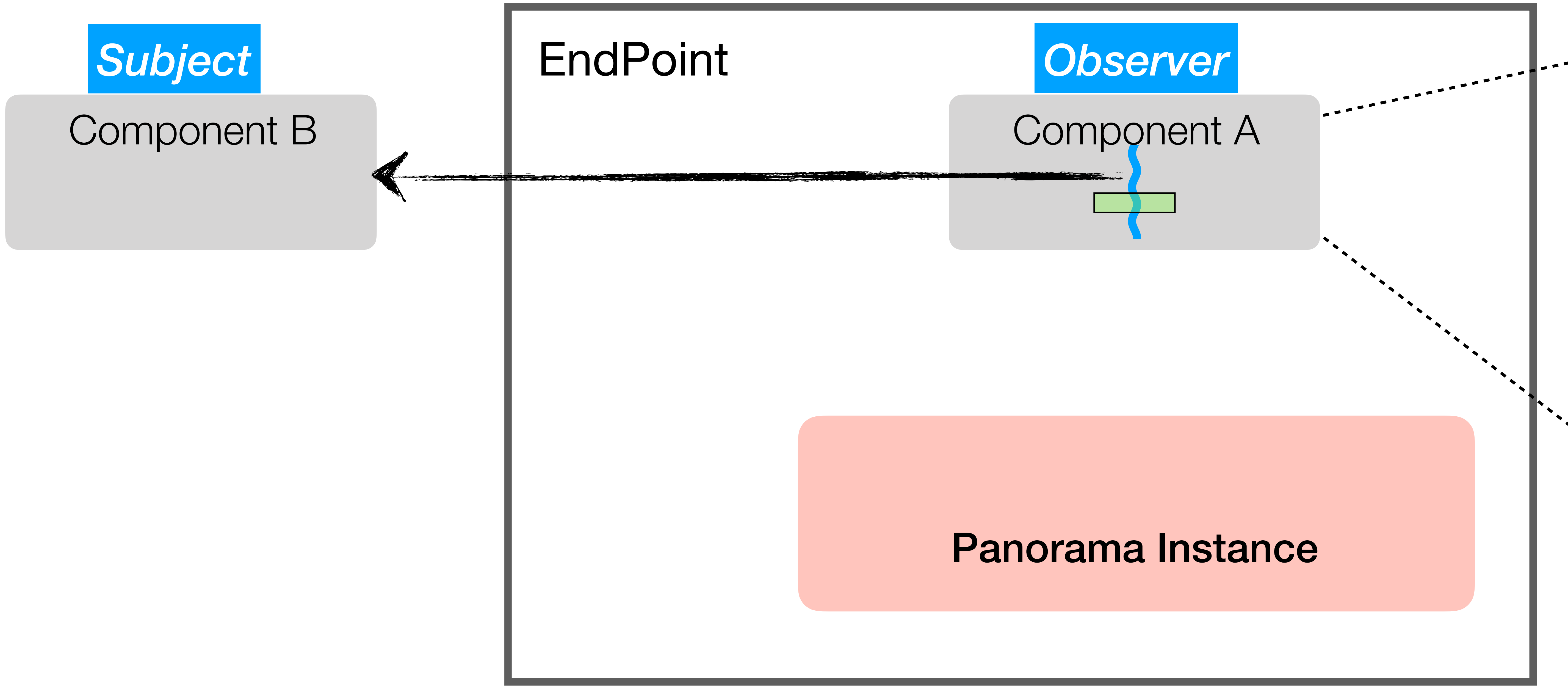
Overview of Panorama - Runtime

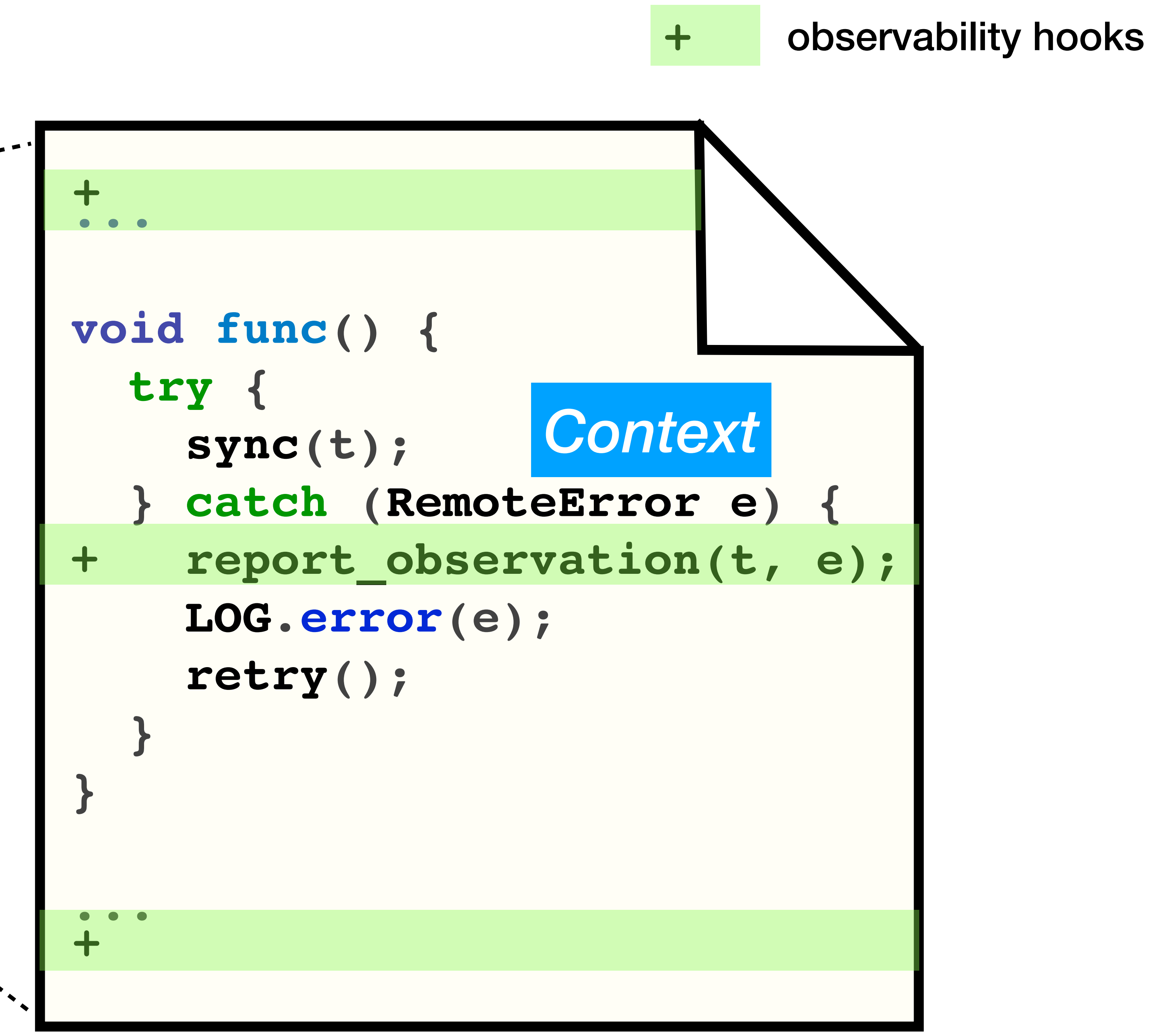


Overview of Panorama - Runtime



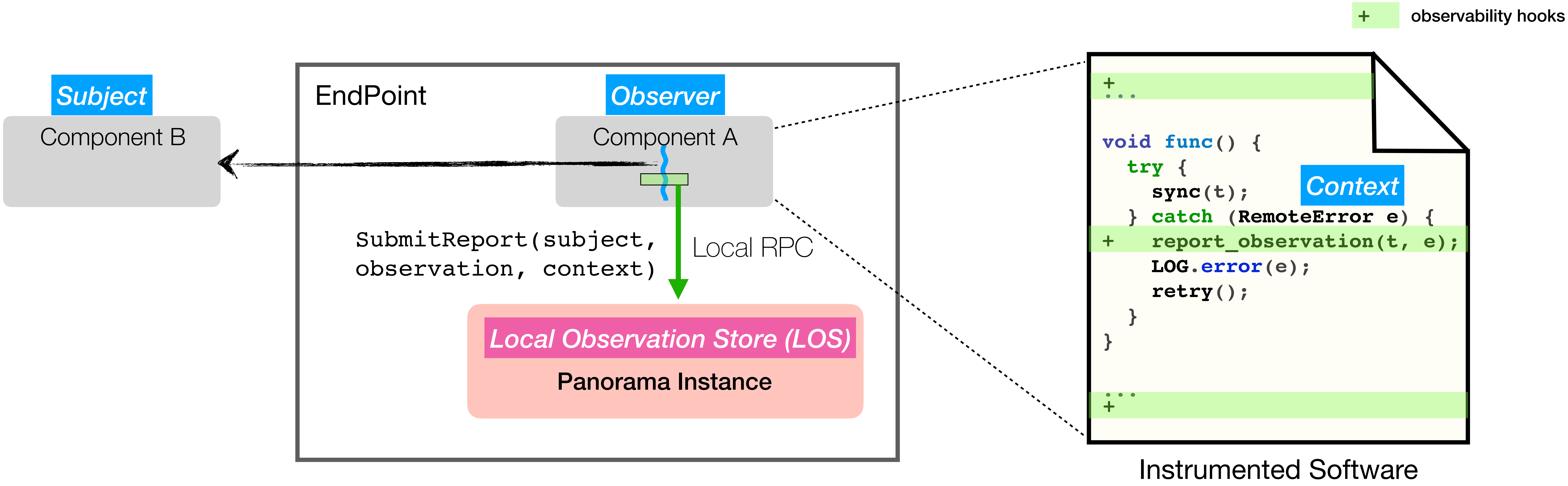
Overview of Panorama - Runtime



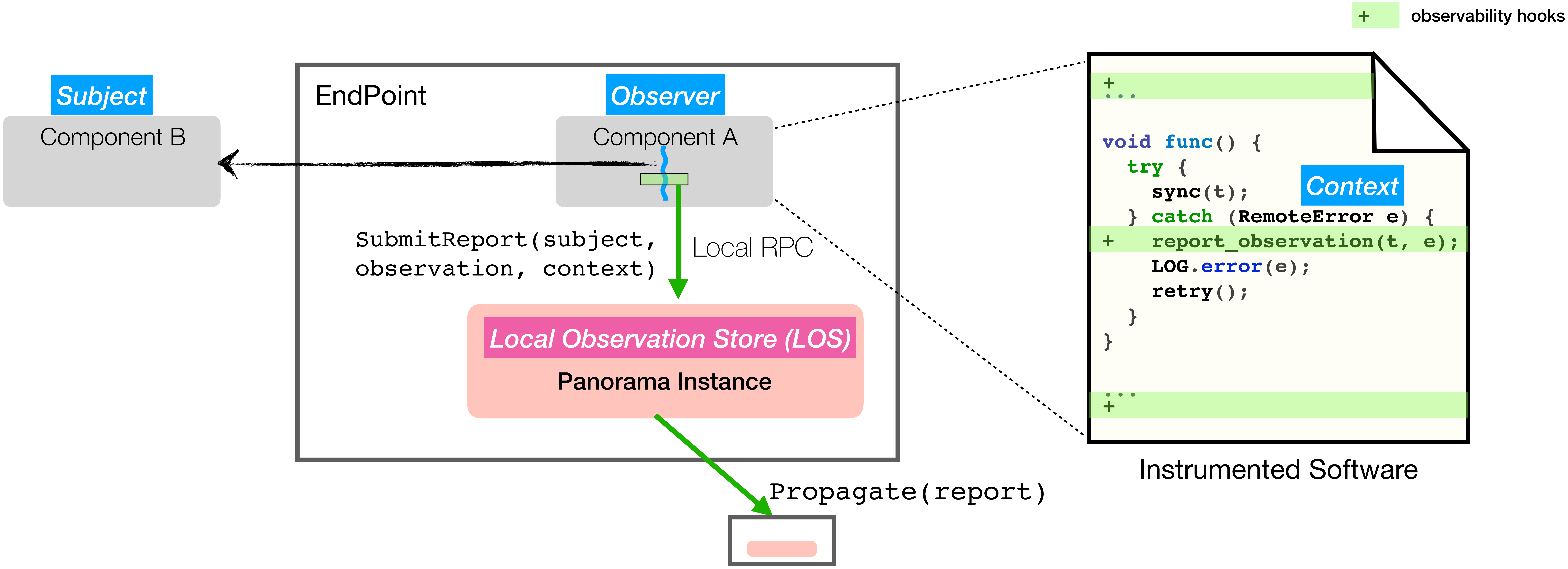


Instrumented Software

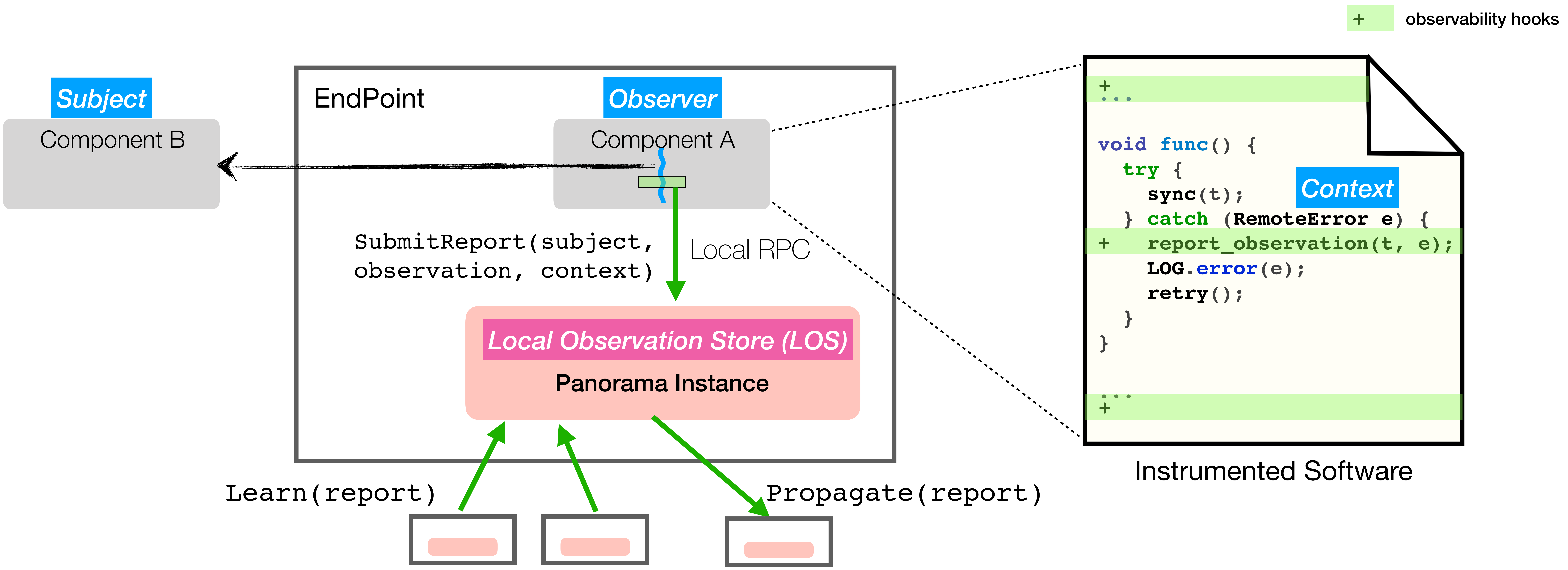
Overview of Panorama - Runtime



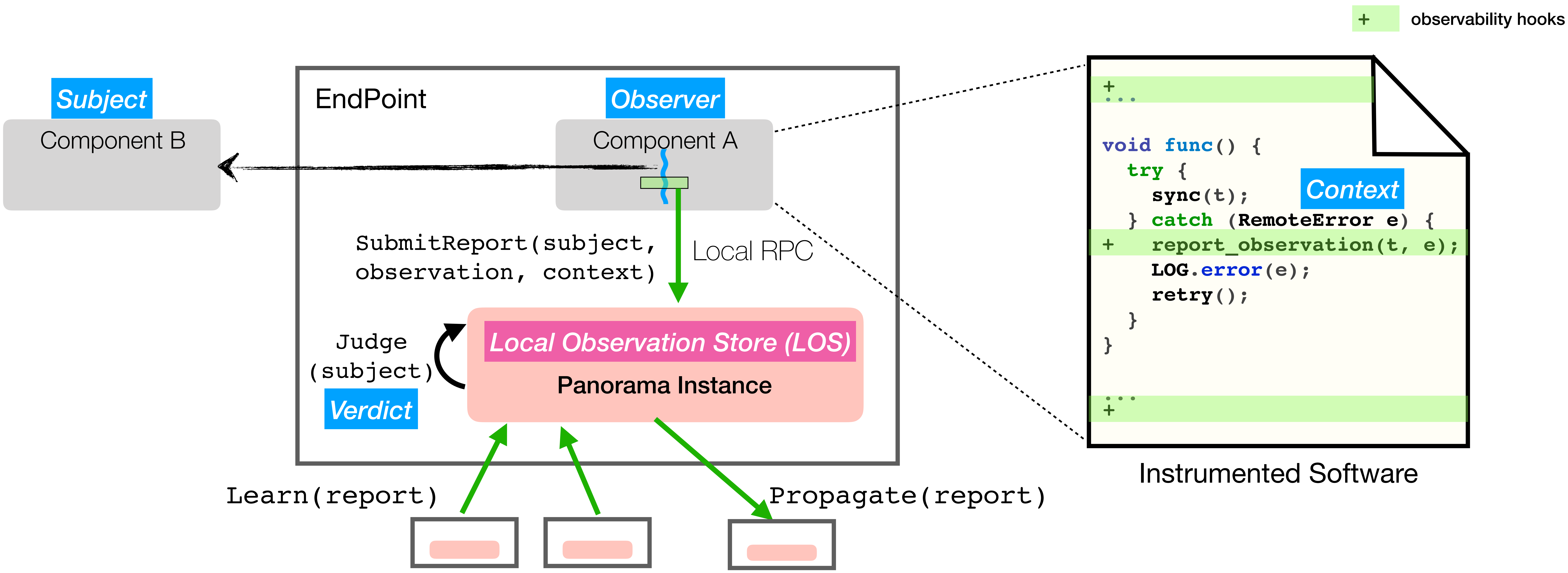
Overview of Panorama - Runtime



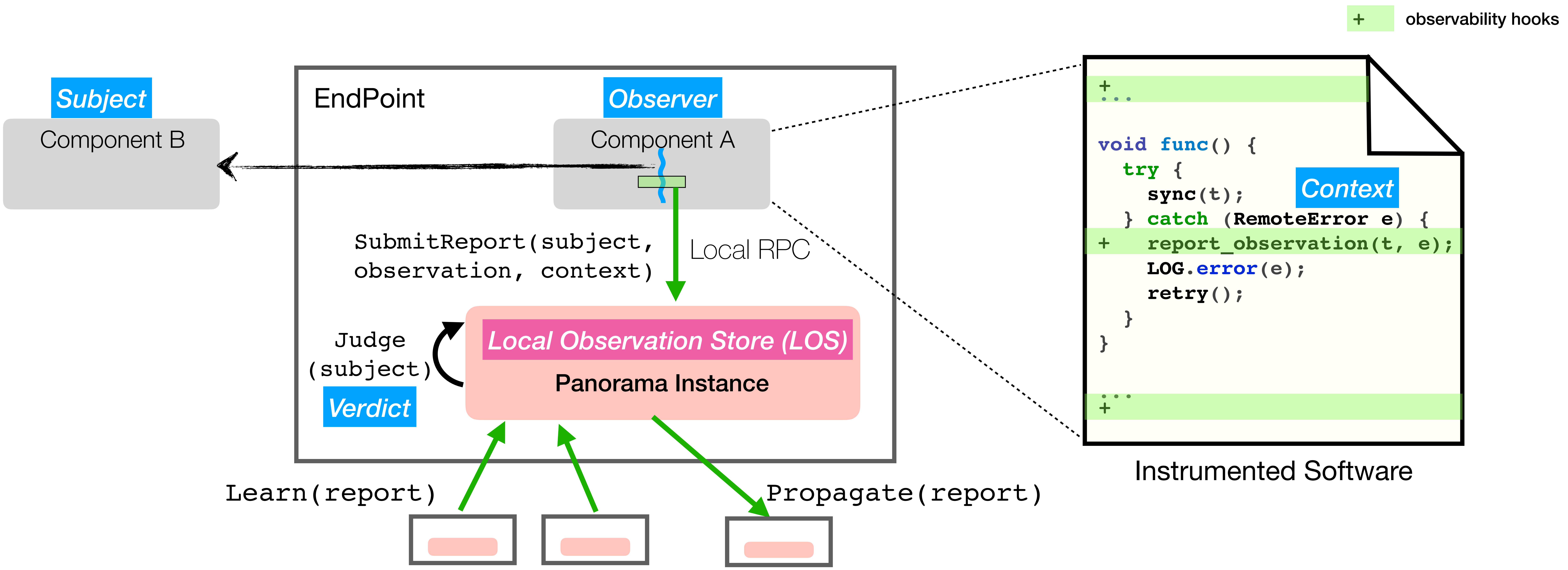
Overview of Panorama - Runtime



Overview of Panorama - Runtime



Overview of Panorama - Runtime



This talk focuses on the analysis stage; see paper for details of the runtime service

Design Challenges

- Observations dispersed in software's source code
- Observation collection may slow down observer's normal service
- Diverse programming paradigms affect observability

Observability Analysis

- Step ❶ : locate domain-crossing invocations (*ob-boundary*)
- Step ❷ : identify observer and observed
- Step ❸ : extract observation (*ob-point*)

Example of Observability Analysis

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

Example of Observability Analysis

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

1 locate ob-boundary

Example of Observability Analysis

❷ identify observer and subject

❶ locate ob-boundary

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

Example of Observability Analysis

2 identify observer and subject

1 locate ob-boundary

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

Example of Observability Analysis

❷ identify observer and subject

❶ locate ob-boundary

❸ extract observation (ob-point)
(invalid reply)

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

Example of Observability Analysis

❷ identify observer and subject

❶ locate ob-boundary

❸ extract observation (ob-point)
(invalid reply)

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

Example of Observability Analysis

❷ identify observer and subject

```
void deserialize(DataTree dt, InputArchive ia) {
  DataNode node = ia.readRecord("node");
  if (node.parent == null) {
    LOG.error("Missing parent.");
    throw new IOException("Invalid Datatree");
  }
  dt.add(node);
}
void snapshot() {
  InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
  try {
    deserialize(getDataTree(), ia);
  } catch (IOException e) {
    sock.close();
  }
}
```

❶ locate ob-boundary

❸ extract observation (ob-point)
(invalid reply)

❸ extract observation (ob-point)
(no reply)

❶ Locate Observation Boundary

- **Function innovations that cross different components**
  - socket I/O, RPCs, messaging, etc.
    - `java.io.DataOutput.write*()`
    - `org.apache.hadoop.hbase.ipc.RpcCall.sendResponseIfReady()`
    - `protobuf.RegionServerStatusService.*()`

❷ Identify Observer

- **Observer identity is global and tracks the source of observations**
  - The id/name the process uses in the distributed system
    - e.g., `QuorumServer.id` (value from `myid` file in ZooKeeper service config)
- **One-time registration with Panorama instance when the process starts**
  - Panorama builds a map between identity and endpoint address for reverse lookup

❷ Identify Observed (Subject)

- Information is in the argument or object of ob-boundary

lookup subject identity from address

  - `public int sendRR(MessageOut message, InetAddress to, IAsyncCallback cb)`

  - `public NNCommand NNProtocolTranslatorPB.startCheckpoint(Registration registration)`

subject identity in the field

`private NNProtocolPB rpcProxy;`

- Sometimes the information is provided by a proxy

  - `rpc HRegionServer.RegionServerStatusService.*(*)`

subject identity from ZooKeeper

`masterAddressTracker.getMasterAddress(false);   // get master address from ZooKeeper`

❸ Extract Negative Evidence

- Errors/exceptions that originate from the ob-boundary
  - instrument exception handlers
  - needs to distinguish generic exceptions (e.g., `IOException`)
- Unexpected reply content
  - `if (!reply.sig.equals("BenWasHere")) error("Bad signature");`
  - intra-procedural analysis to look for errors that have control-dependency on the reply

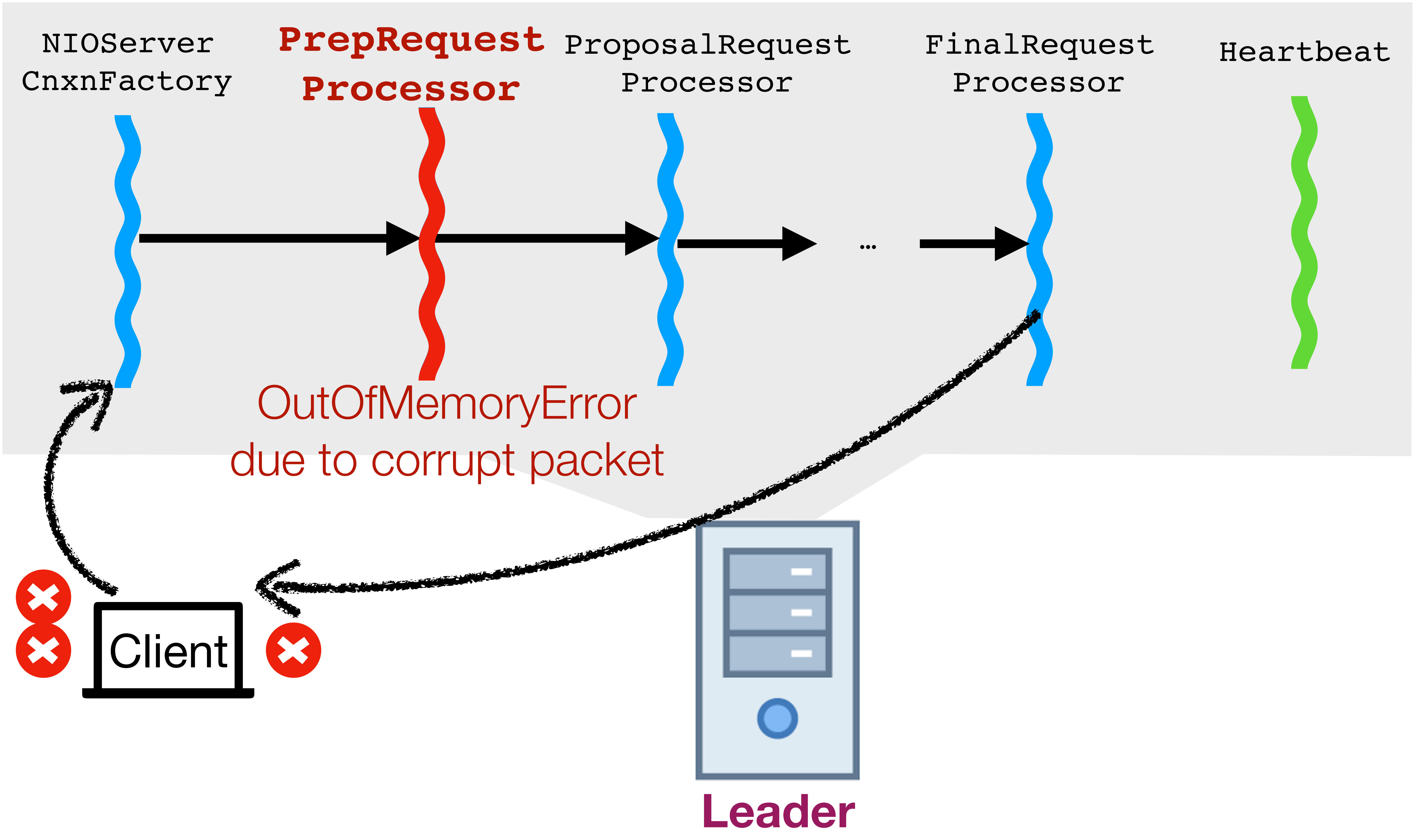
❸ Extract Positive Evidence

- **Successful invocation from ob-boundary**
  - it's OK to submit a positive observation immediately and later find errors in reply content
- **If domain-crossing occurs frequently, positive observations are excessive**
  - coalesce similar positive ob-points that are located close together
  - library buffers frequent observations and sends them as one aggregate observation

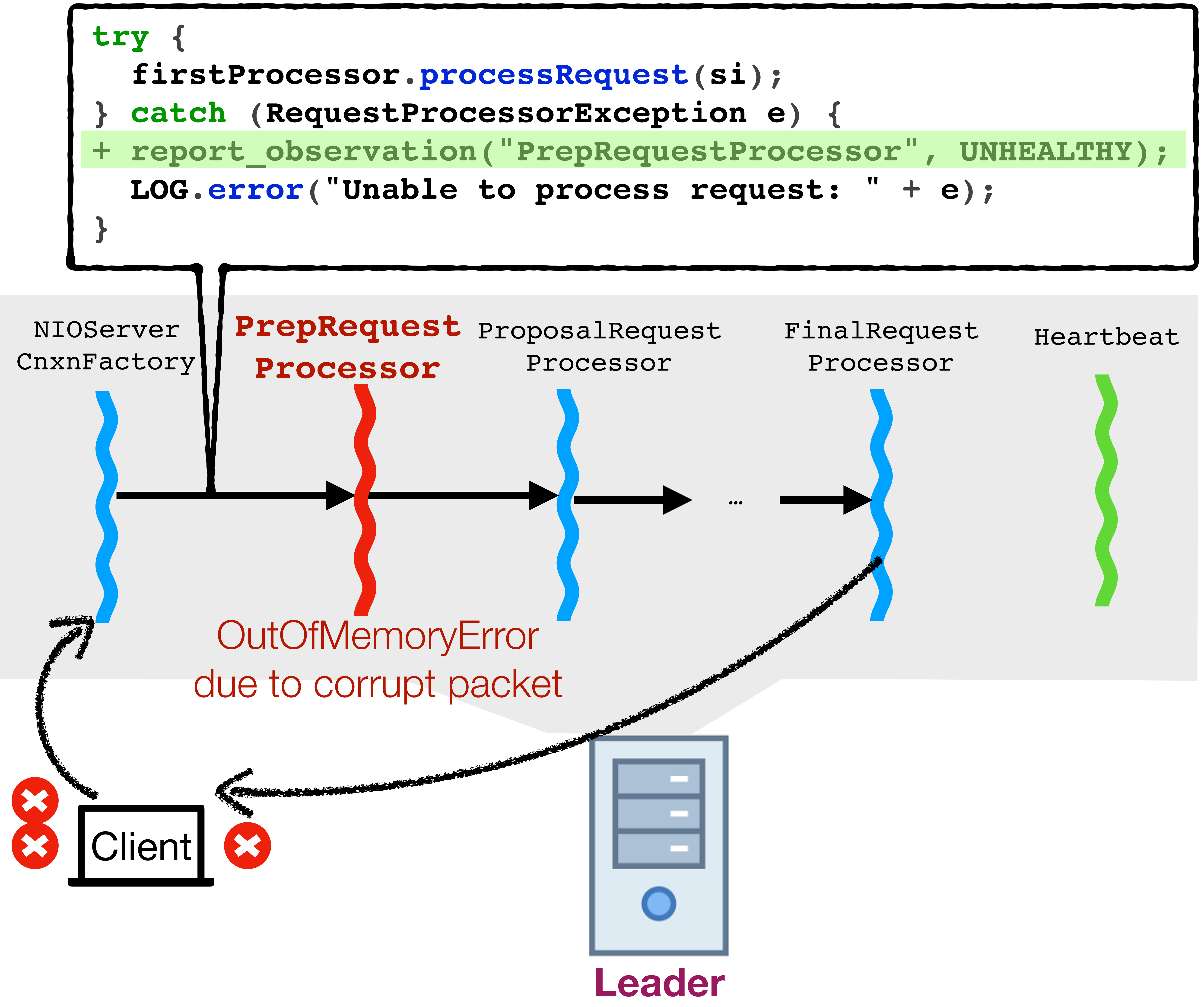
Design Challenges

- Observations dispersed in software’s source code
- Observation collection may slow down observer’s normal functionality
- Diverse programming paradigms affect observability

Learning from a Subtle Case



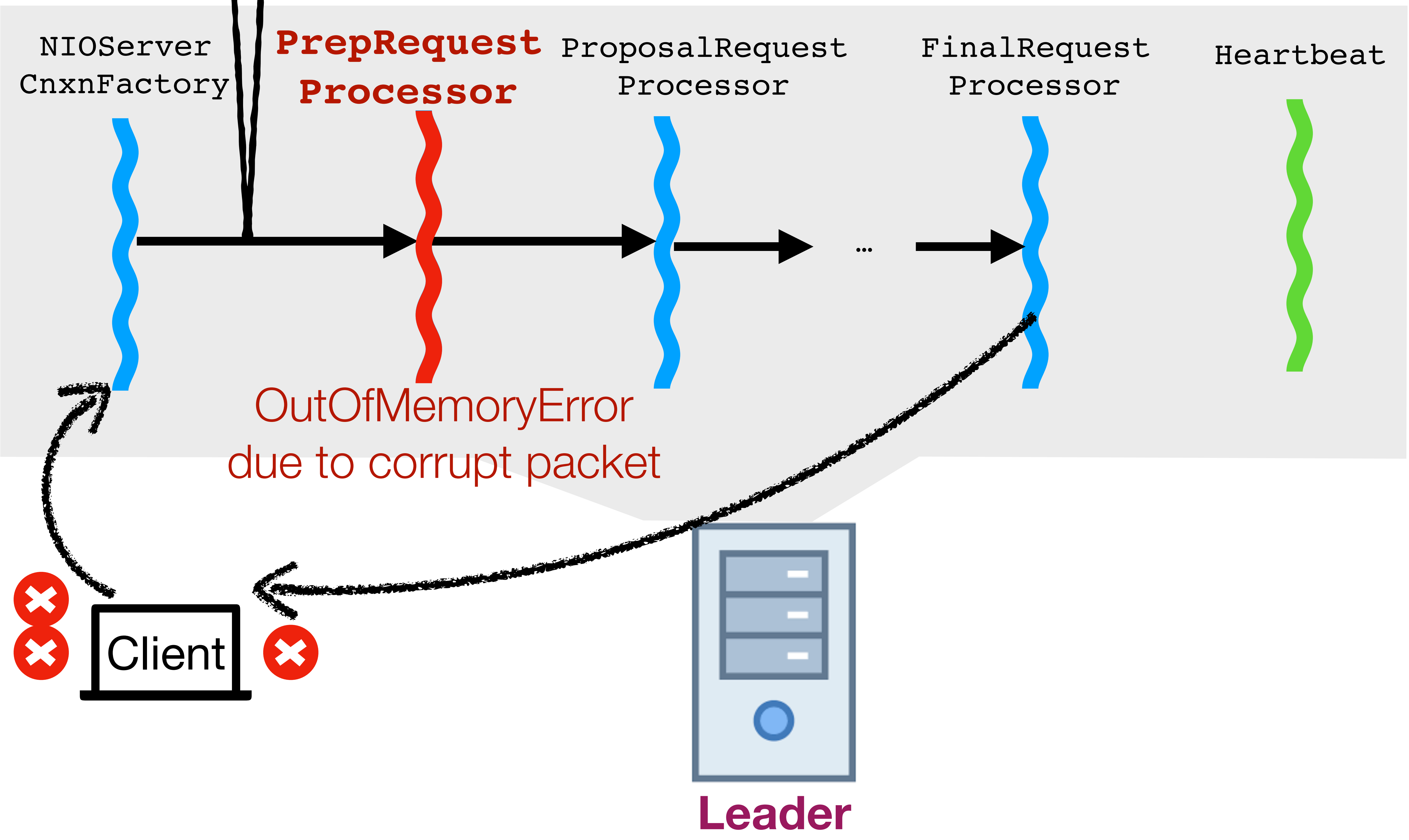
Learning from a Subtle Case



Learning from a Subtle Case

```
try {
  firstProcessor.processRequest(si);
} catch (RequestProcessorException e) {
+ report_observation("PrepRequestProcessor", UNHEALTHY);
  LOG.error("Unable to process request: " + e);
}
```

PrepRequest Processor

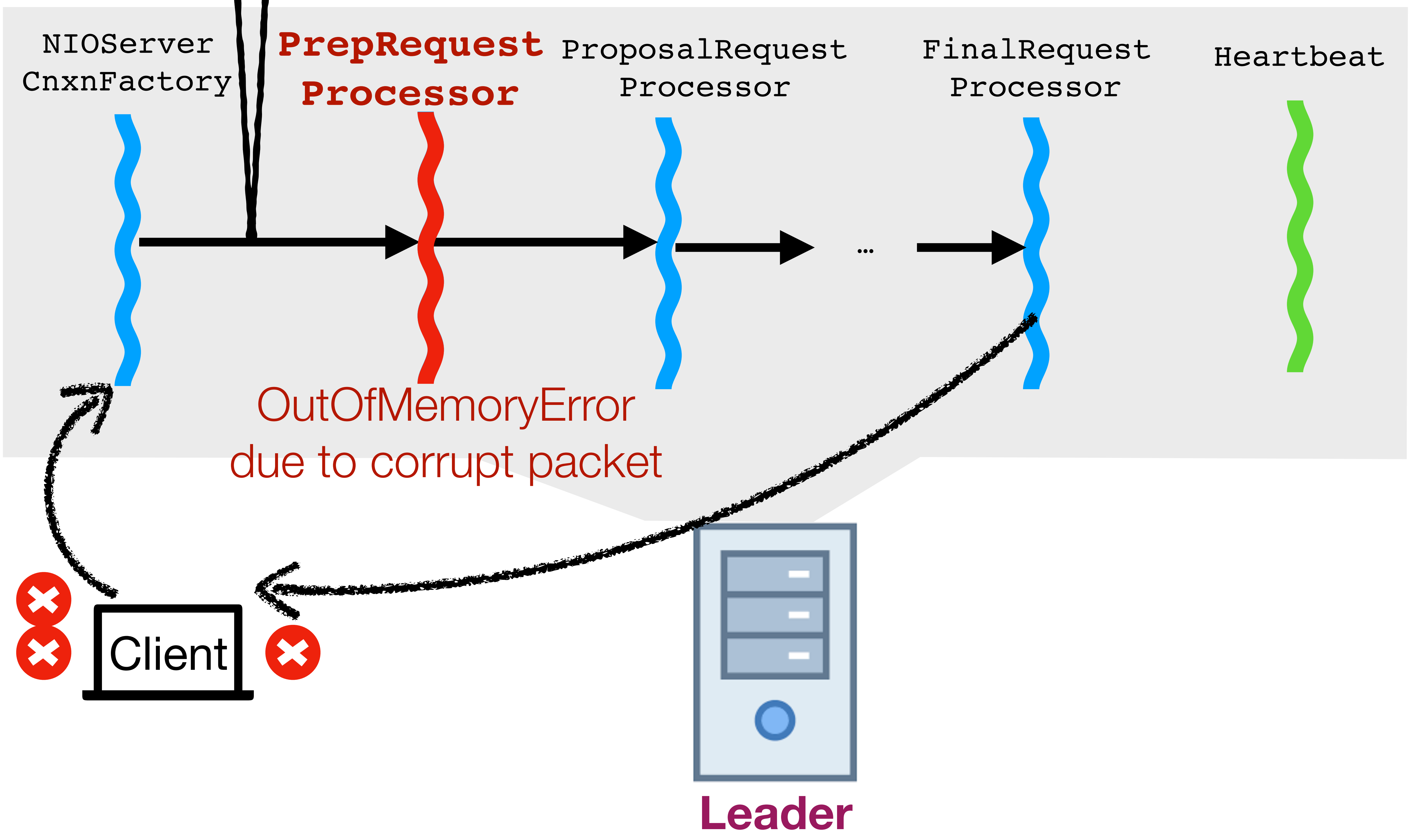


Learning from a Subtle Case

```
try {
  firstProcessor.processRequest(si);
} catch (RequestProcessorException e) {
+ report_observation("PrepRequestProcessor", UNHEALTHY);
  LOG.error("Unable to process request: " + e);
}
```

```
public class PrepRequestProcessor extends Thread {
  BlockingQueue<Request> requests = new ...
  void run() {
    while (true) {
      Request r = requests.take();
      pRequest(r); // the OOM occurred inside
    }
  }
  public void processRequest(Request request)
      throws RequestProcessorException {
    requests.add(request);
  }
}
```

PrepRequest Processor

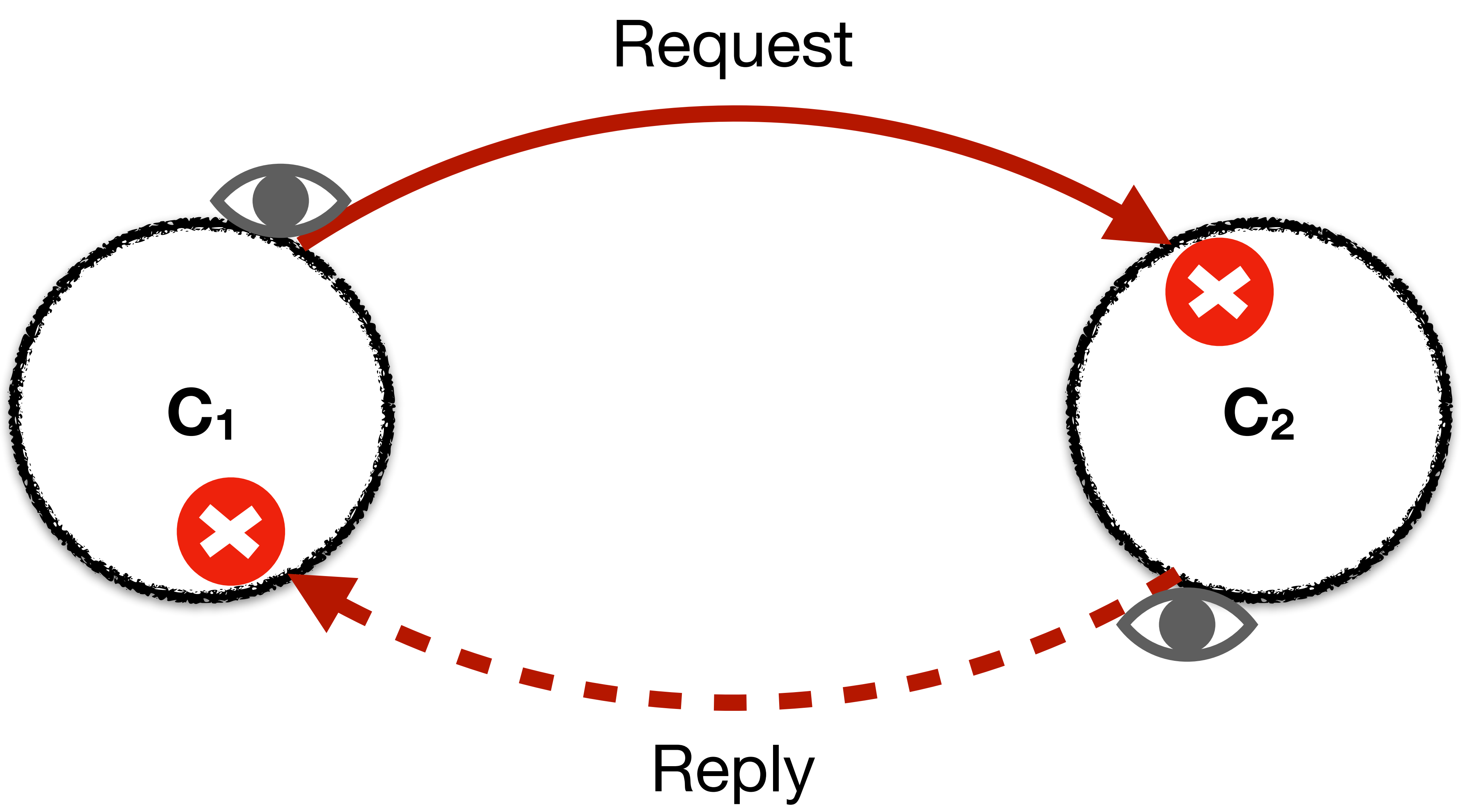


Learning from a Subtle Case

- Indirection can *reduce* failure observability

- Evidence is made about the indirection layer instead of actual component

Observability Design Patterns

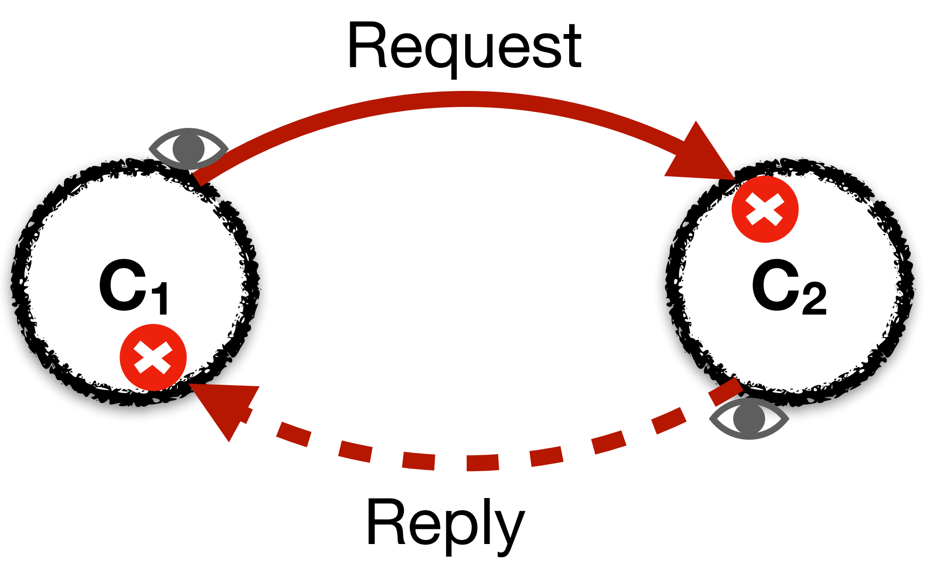


C2's failure is observable to C1

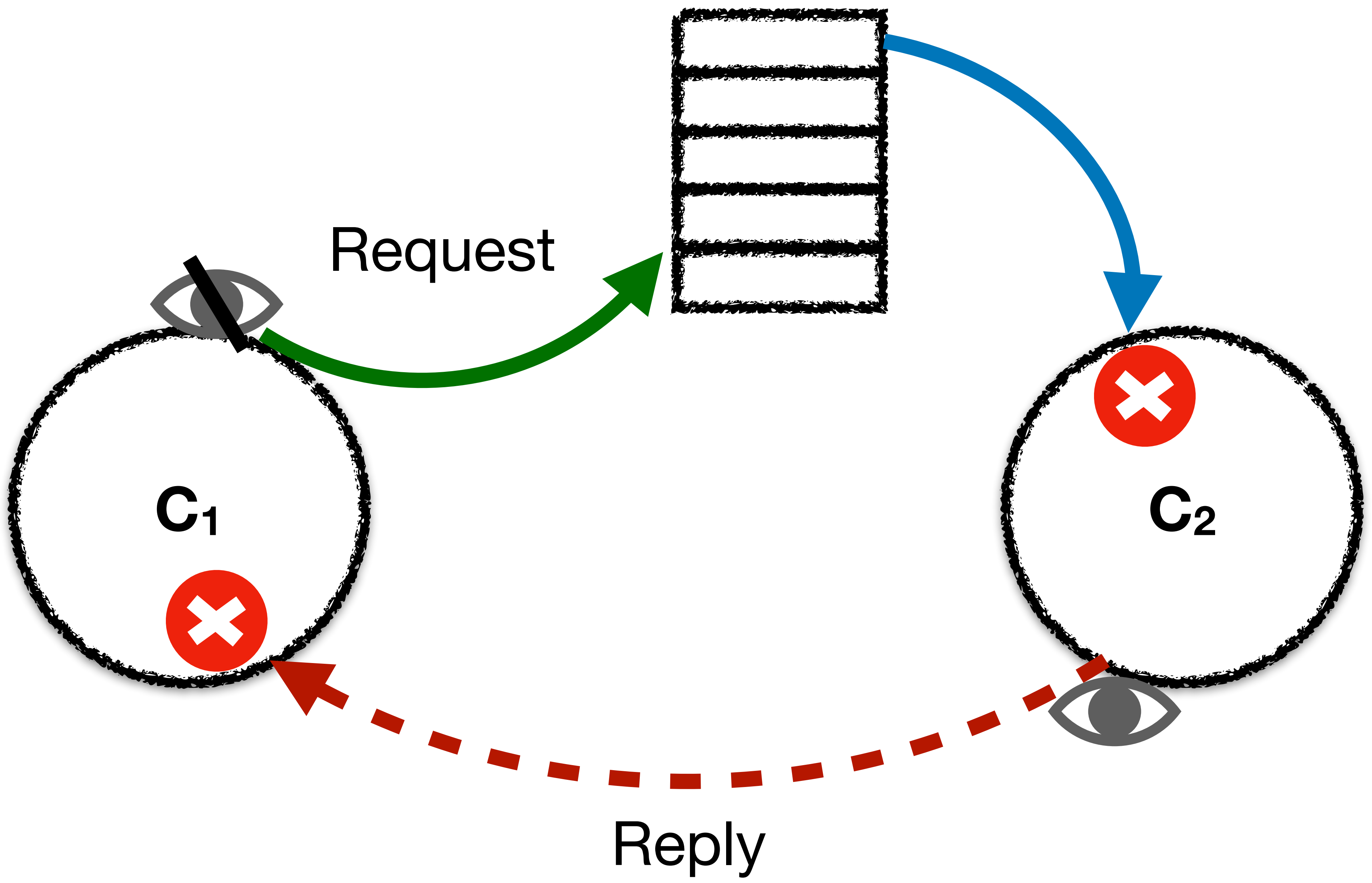
C1's failure is observable to C2

Interaction of two components C_1 and C_2

Observability Design Patterns



No indirection

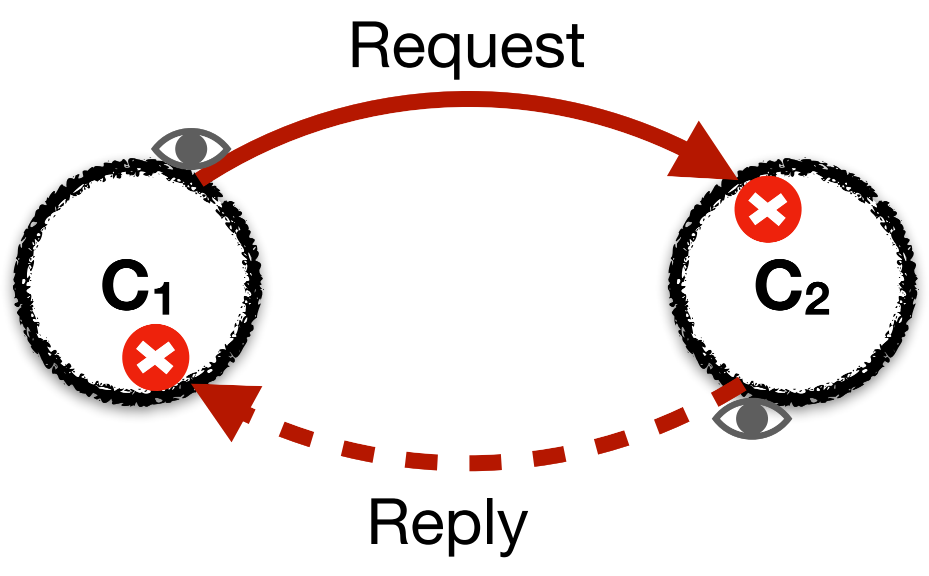


C2's failure is unobservable to C1

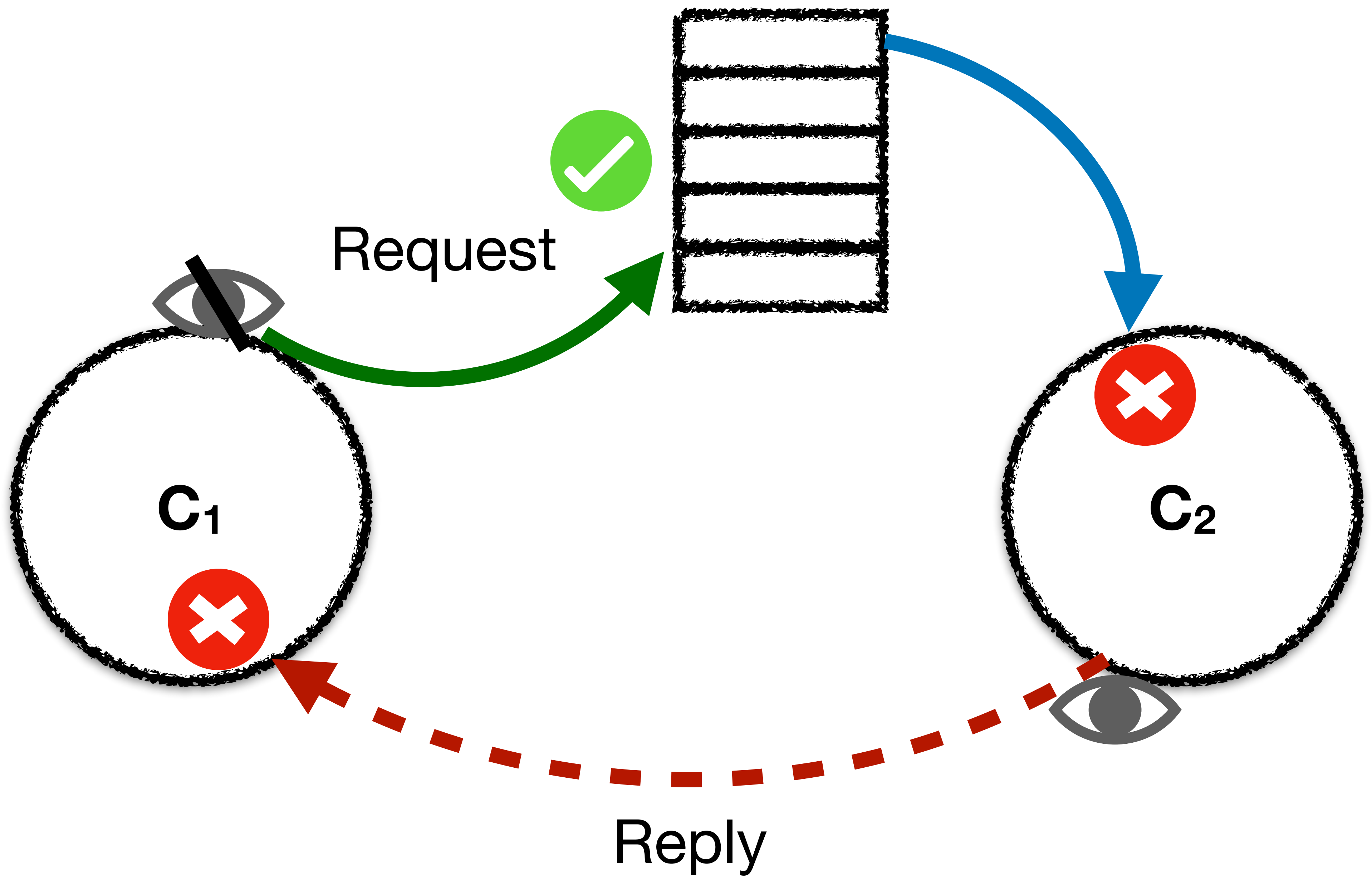
C1's failure is observable to C2

Interaction of two components C_1 and C_2

Observability Design Patterns



No indirection

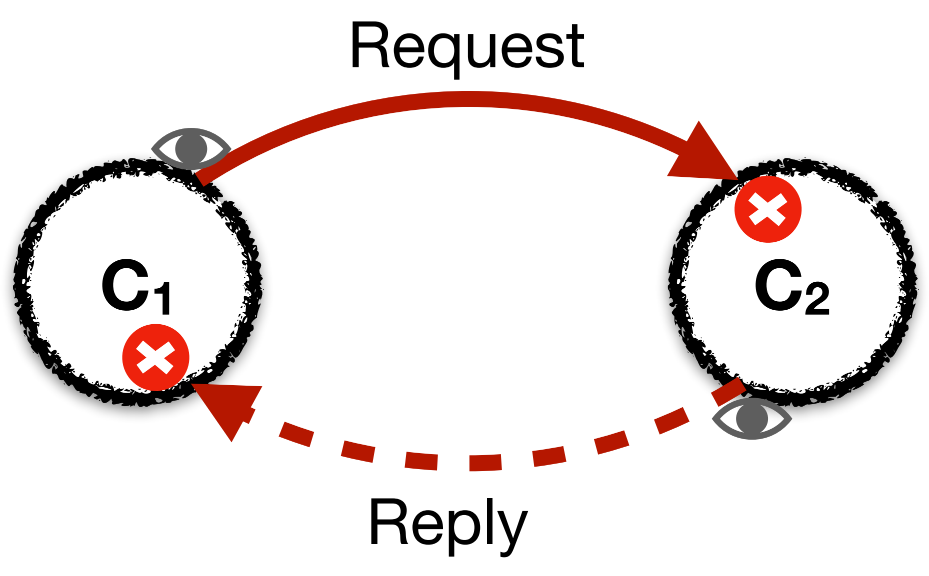


C2's failure is unobservable to C1

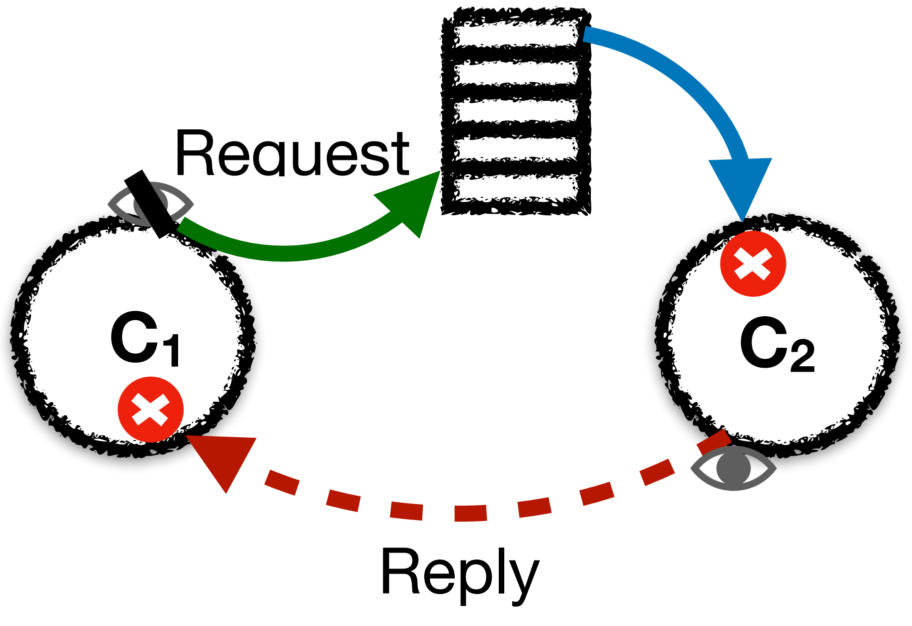
C1's failure is observable to C2

Interaction of two components C_1 and C_2

Observability Design Patterns



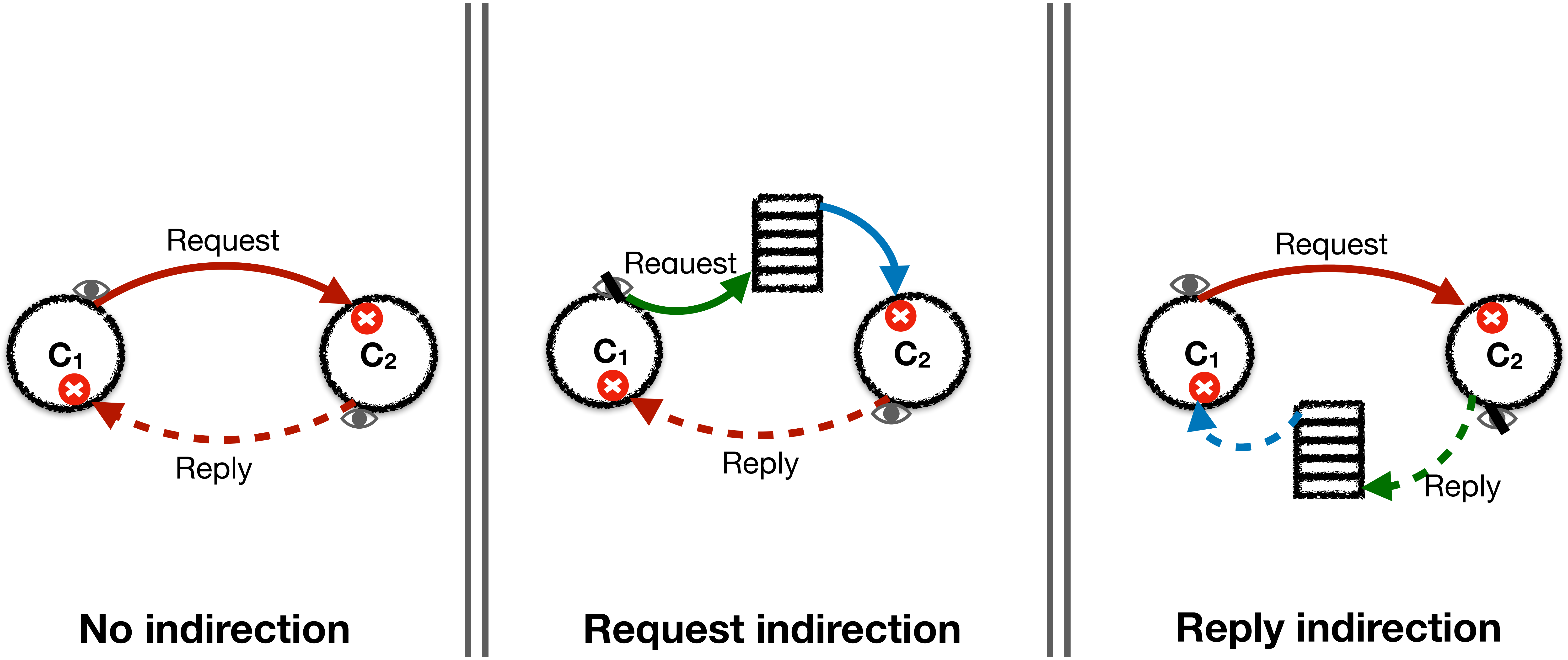
No indirection



Request indirection

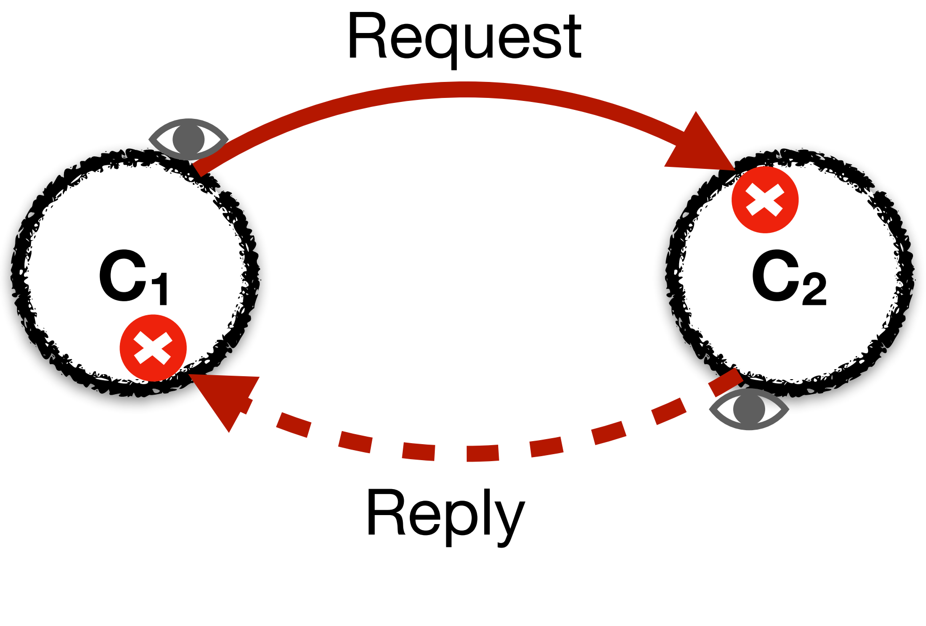
Interaction of two components C_1 and C_2

Observability Design Patterns

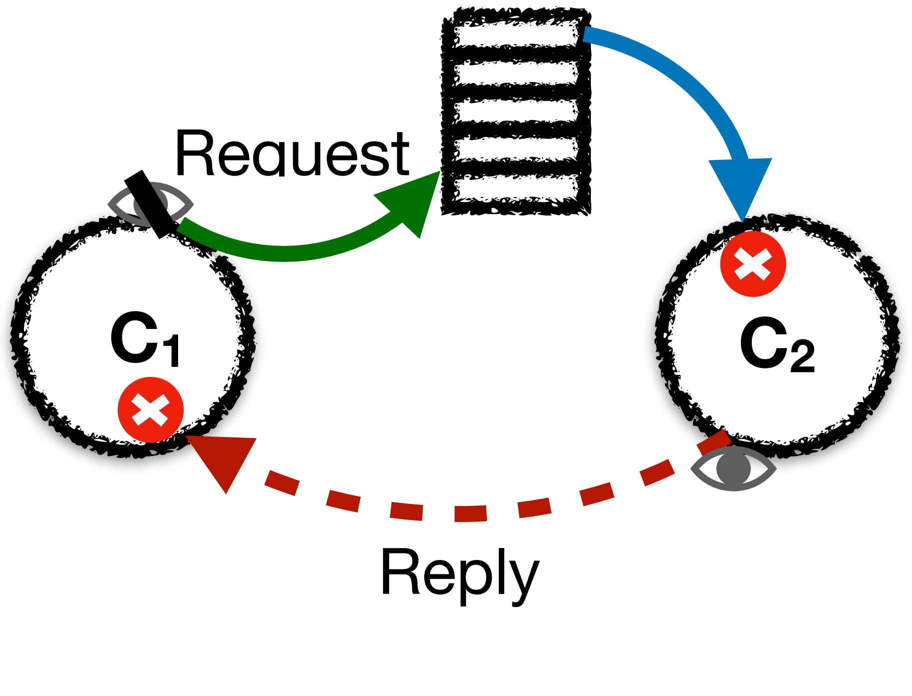


Interaction of two components C_1 and C_2

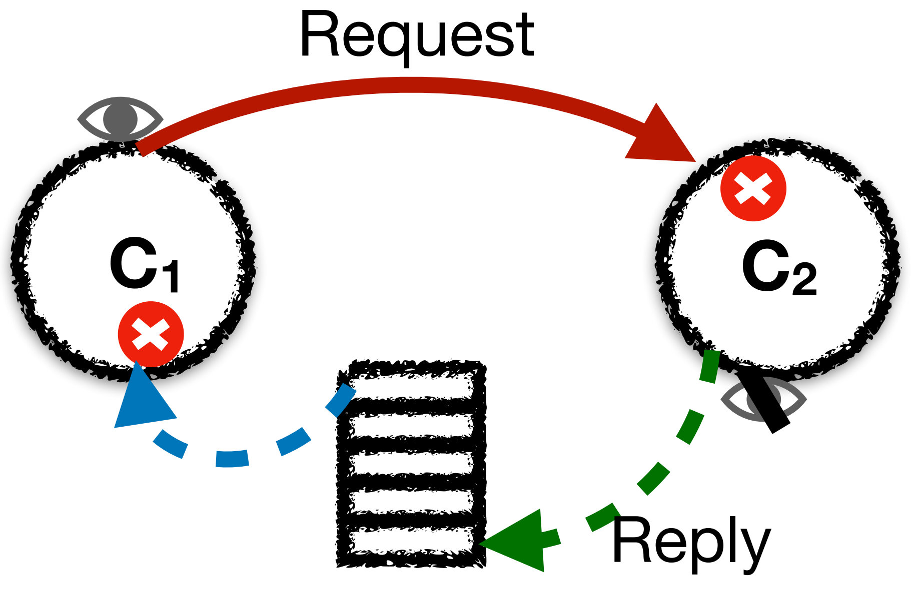
Observability Design Patterns



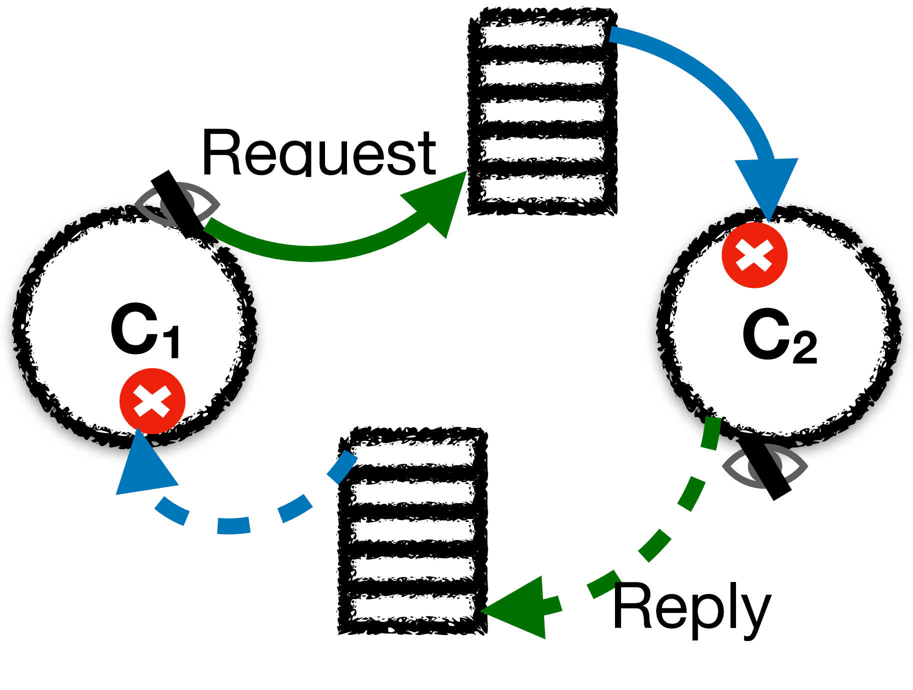
No indirection



Request indirection



Reply indirection



Full indirection

Interaction of two components C_1 and C_2

Capture Observation w/ Indirection

- **Fundamental issue is that an observation becomes split**
  - We call them `ob-origin` and `ob-sink`

positive, but temporary and weak evidence

complete and strong evidence

Identify Ob-Origin and Ob-Sink

- **Ob-origin is ob-boundary in asynchronous method**
  - Submit a `PENDING` observation after ob-origin
- **Ob-sink lies in notification mechanism**
  - (1) callback set invocation; (2) blocking wait callback get; (3) checking a completion flag
  - Submit a `HEALTHY/UNHEALTHY` observation before ob-sink (1) or after ob-sink (2), (3)

Match Ob-Origin and Ob-Sink

- Match based on `<subject, context, [request_id]>`
- A later `HEALTHY` observation merges a past `PENDING` observation
  - the positive observation is now complete
- A later `UNHEALTHY` observation overrides a past `PENDING` observation
- If `PENDING` observations outstanding for too long, degrade to `UNHEALTHY`

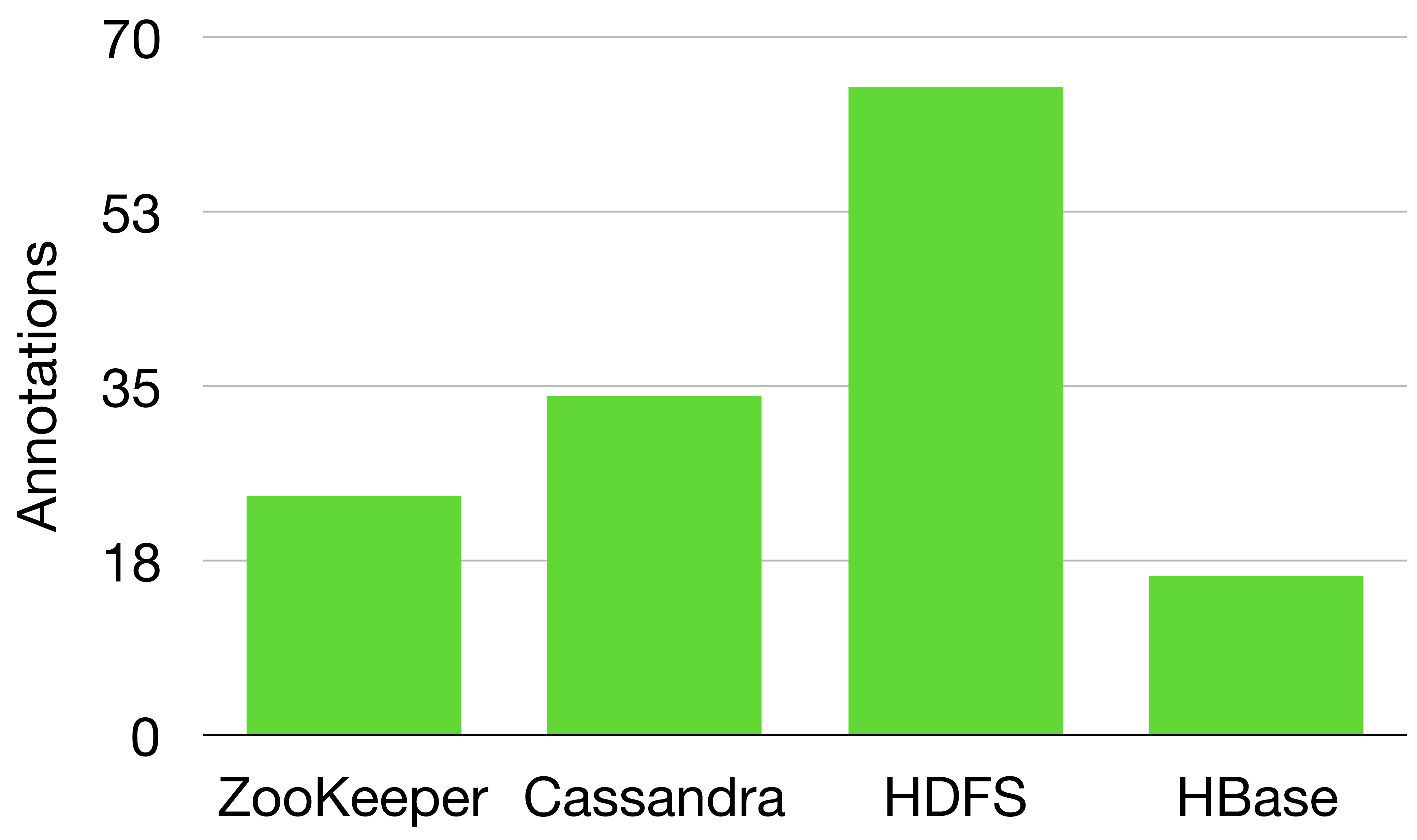
Implementation

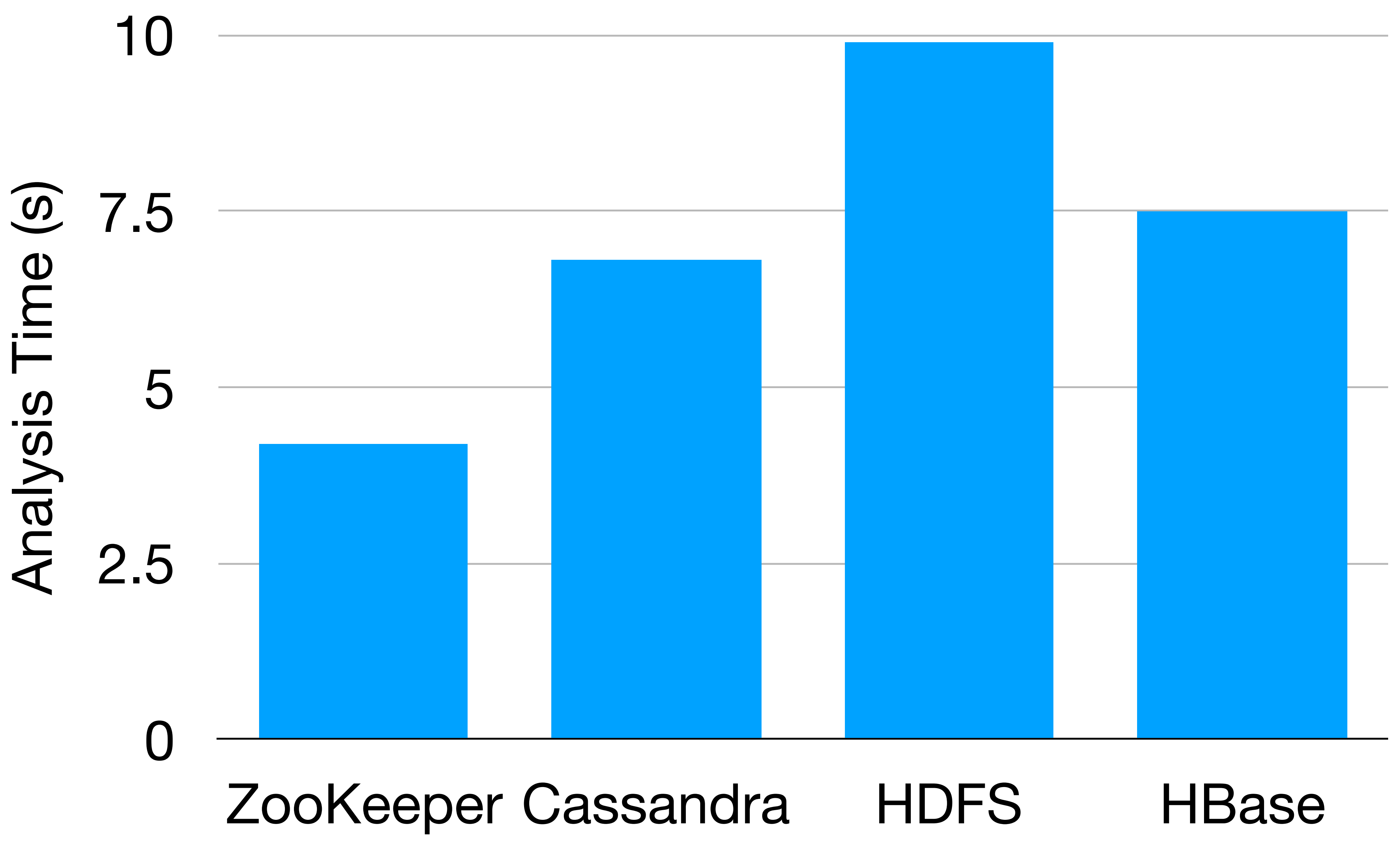
- Panorama system implemented in Go + gRPC
    - Easy to plug-in with different clients
- A thin library provides async reporting, buffering, etc.
    - Most reporting does not directly trigger local RPCs
- Panorama analyzer implemented with Soot and AspectJ

Evaluation

Analysis	Detection Service		
Integration with real-world systems	Catching crash and gray failures	Reacting to transient failures	Cost for observers
1. Practicality and Effort	**2. Detection Time**	**3. Accuracy**	**4. Overhead**

Integration with Real-world Systems

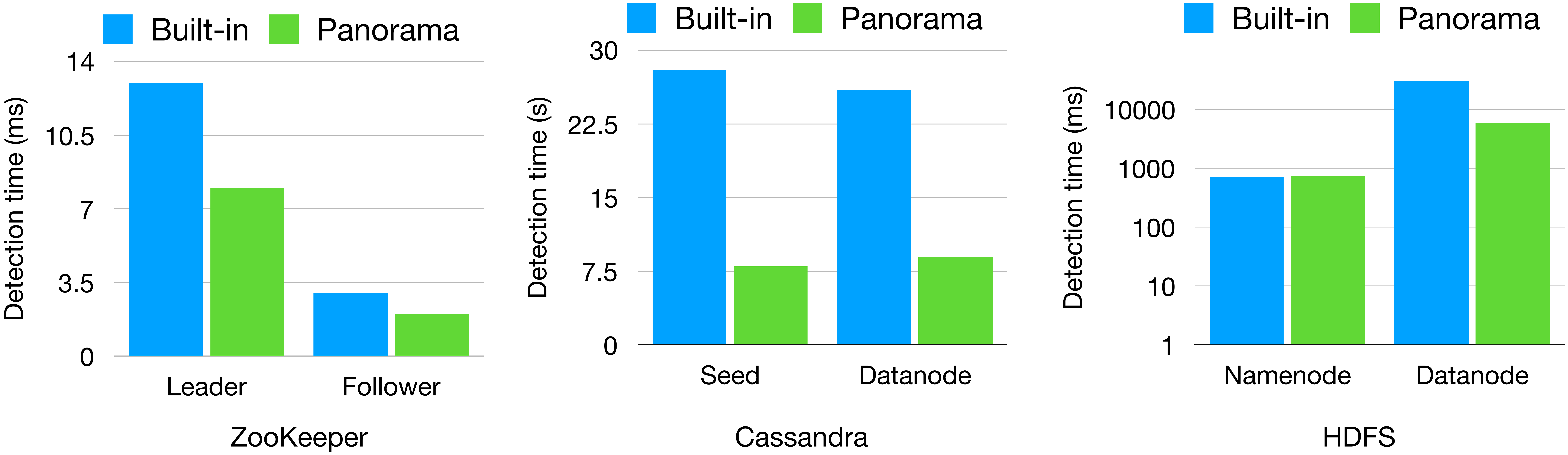




Information about custom interfaces and async mechanisms. Most of them are reusable across different versions.

Analysis and instrumentation are fast.

Detect Injected Crash Failures



Built-in crash failure detectors have to be conservative to deal with asynchrony.

Real-World Gray Failures

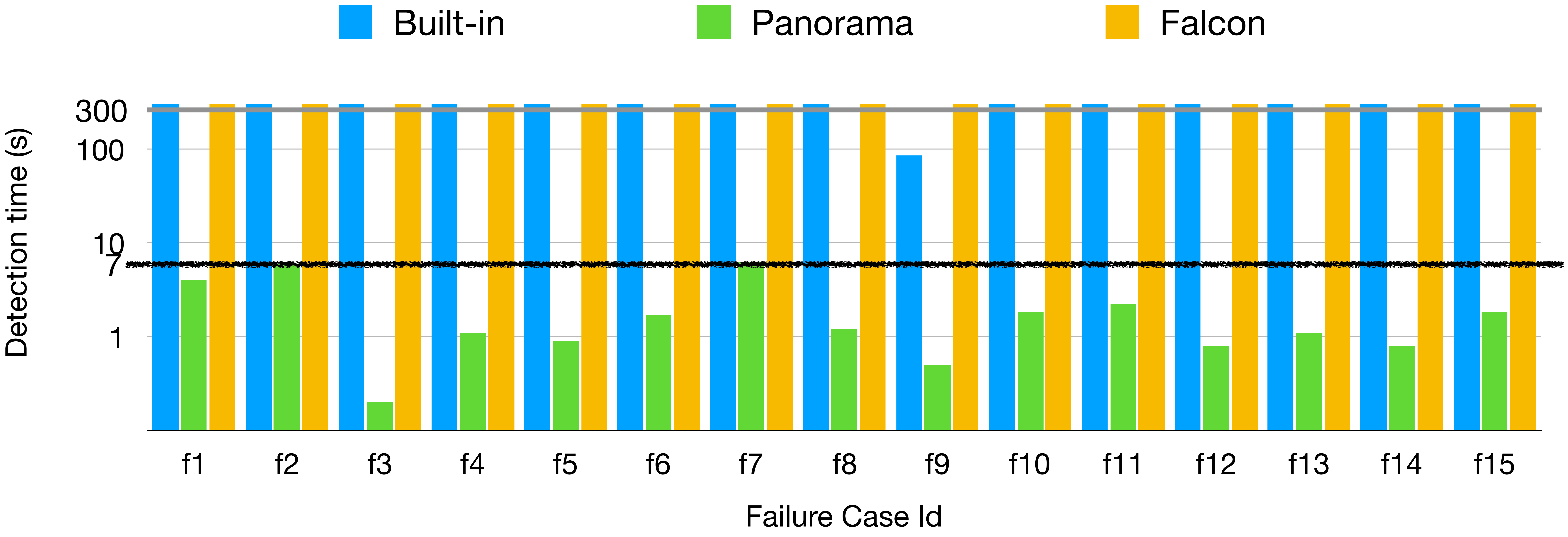
ID	System	Fault Synopsis
f_1	ZooKeeper	faulty disk in leader causes cluster lock-up
f_2	ZooKeeper	transient network partition leads to pro-longed failures in serving
f_3	ZooKeeper	corrupted packet in de-serialization
f_4	ZooKeeper	transaction thread exception
f_5	ZooKeeper	leader fails to write transaction log
f_6	Cassandra	response drop blocks repair operations
f_7	Cassandra	stale data in leads to wrong node states
f_8	Cassandra	streaming silently fail on unexpected error
f_9	Cassandra	commitlog executor exit causes GC storm
f_{10}	HDFS	thread pool exhaustion in master
f_{11}	HDFS	failed pipeline creation prevents recovery
f_{12}	HDFS	short circuit reads blocked due to death of domain socket
f_{13}	HDFS	blockpool fails to initialize but continues
f_{14}	HBase	dead root drive on region server
f_{15}	HBase	replication stalls with empty WAL files

Real-World Gray Failures

ID	System	Fault Synopsis
f_1	ZooKeeper	faulty disk in leader causes cluster lock-up
f_2	ZooKeeper	transient network partition leads to pro-longed failures in serving
f_3	ZooKeeper	corrupted packet in de-serialization
f_4	ZooKeeper	transaction thread exception
f_5	ZooKeeper	leader fails to write transaction log
f_6	Cassandra	response drop blocks repair operations
f_7	Cassandra	stale data in leads to wrong node states
f_8	Cassandra	streaming silently fail on unexpected error
f_9	Cassandra	commitlog executor exit causes GC storm
f_{10}	HDFS	thread pool exhaustion in master
f_{11}	HDFS	failed pipeline creation prevents recovery
f_{12}	HDFS	short circuit reads blocked due to death of domain socket
f_{13}	HDFS	blockpool fails to initialize but continues
f_{14}	HBase	dead root drive on region server
f_{15}	HBase	replication stalls with empty WAL files

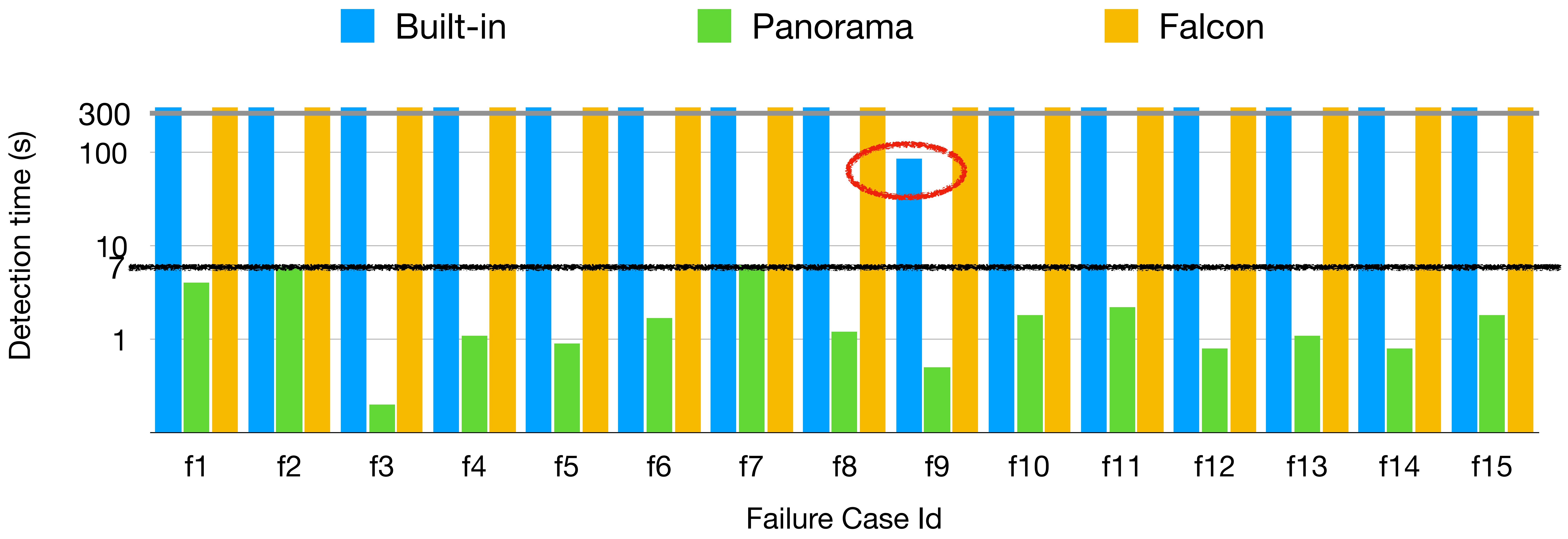
In all cases, severe service disruption occurred (e.g., create requests failed) while the failing component was perceived to be healthy.

Detection Time of Gray Failures



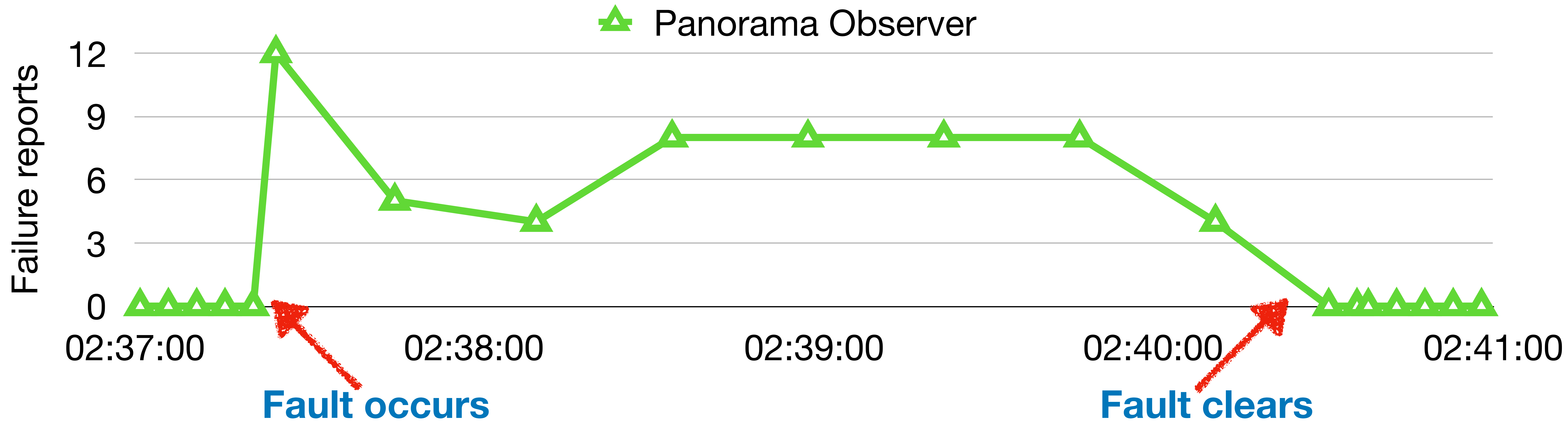
Panorama detects all 15 failures in under 7 seconds; Built-in detectors only detect one case within 300 seconds.

Detection Time of Gray Failures

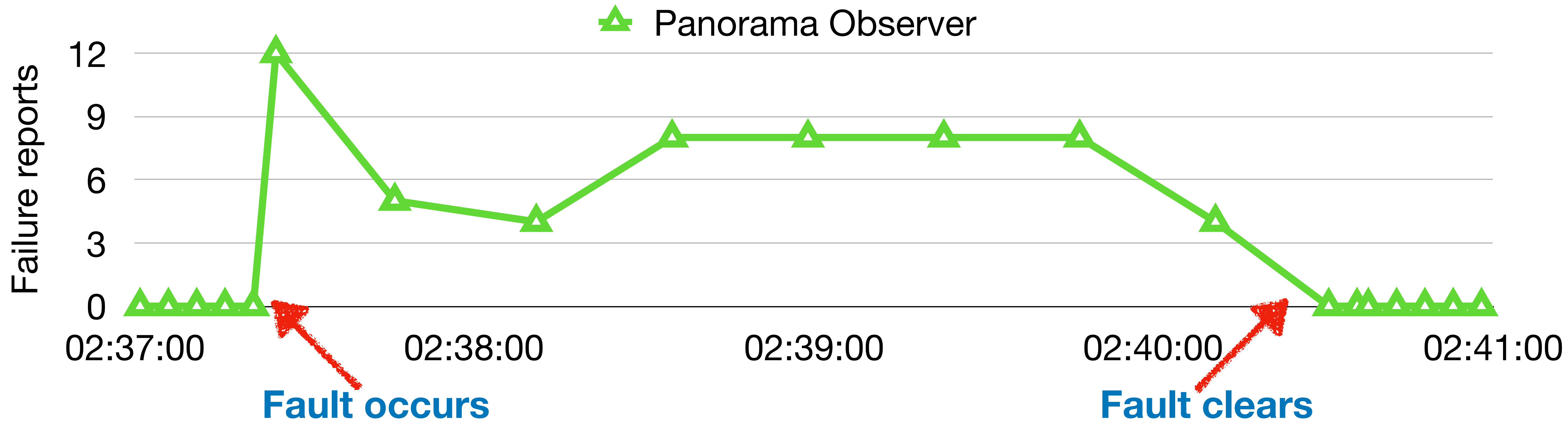


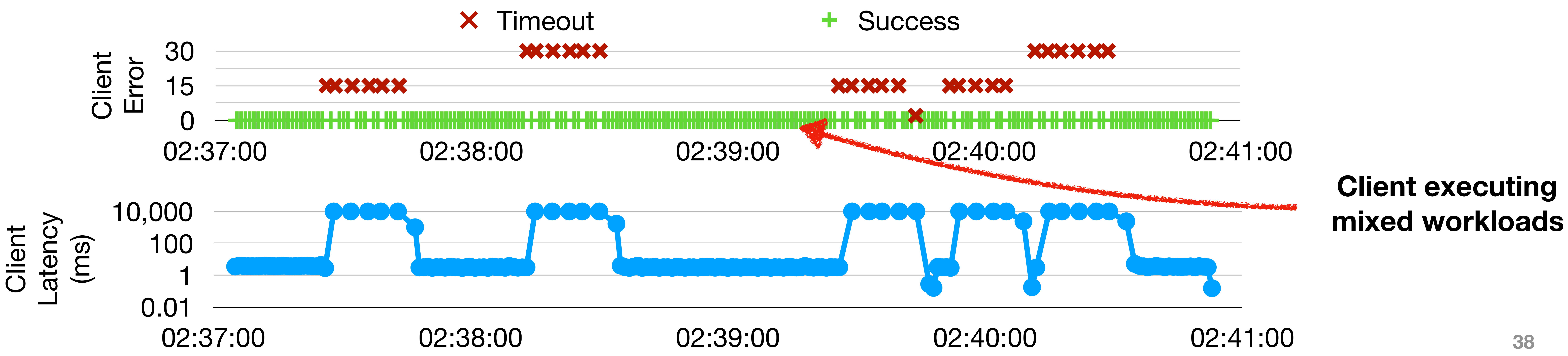
Panorama detects all 15 failures in under 7 seconds; Built-in detectors only detect one case within 300 seconds.

Timeline in Detecting Gray Failure f_1

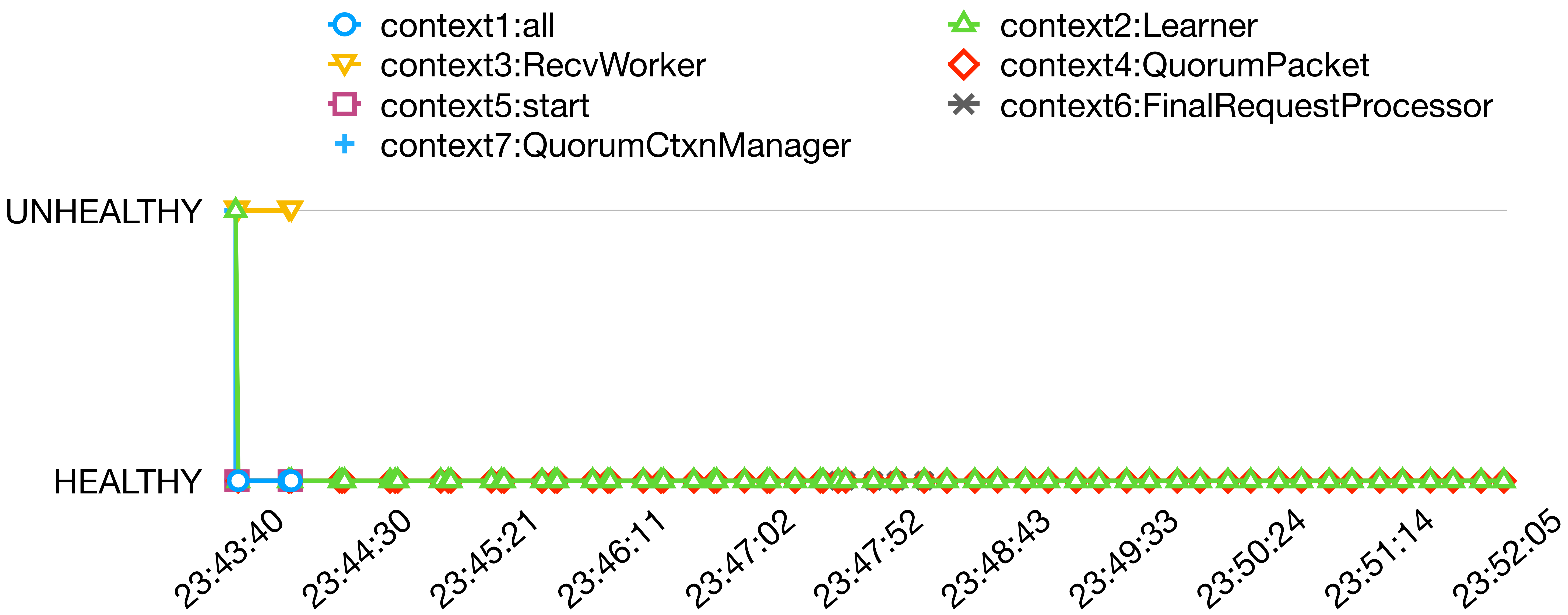


Timeline in Detecting Gray Failure f_1

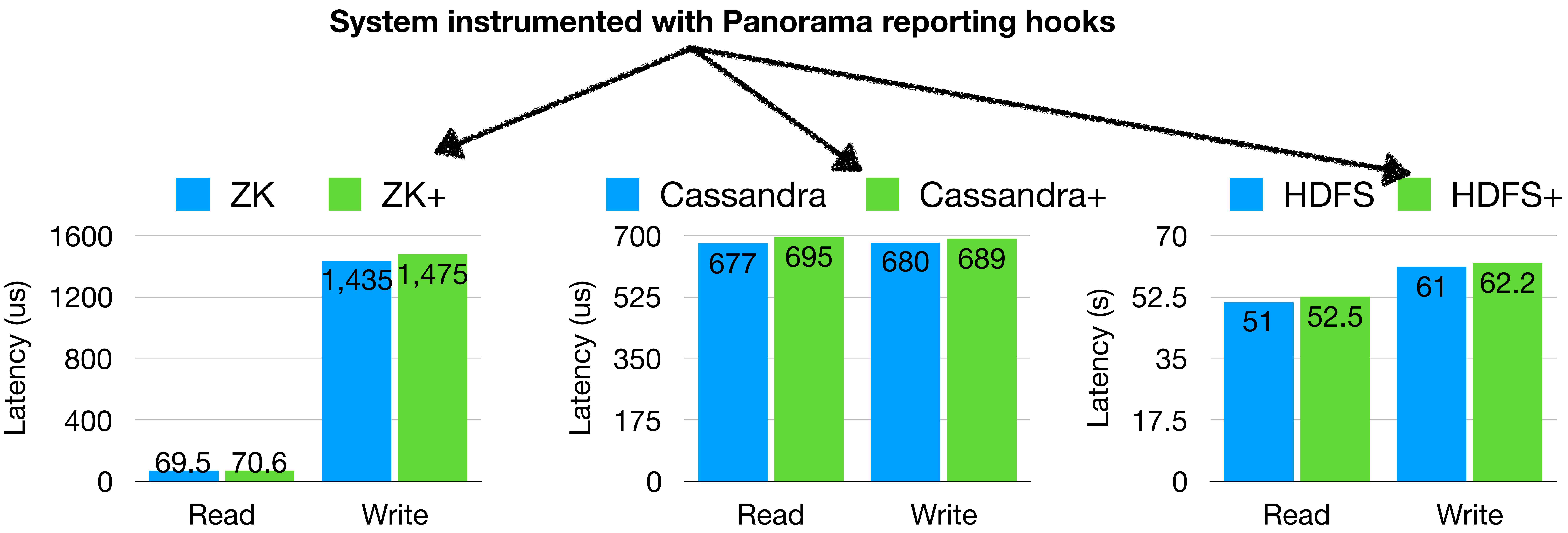




Transient Failures



Latency Overhead to Observers



Less than 3% latency and throughput (not shown) overhead

Limitations & Future Work

- **Panorama relies on dependencies and interactions in large systems**
  - if observability is inherently low, have to improve the built-in FDs.
- **Panorama currently focuses on failure detection**
  - need to integrate detection results to the subject to take actions
- **Panorama currently does not identify root cause of observations**
  - useful for localizing failing component in cascading failures

Related Work

- **Failure Detection**
  - Gossip [Middleware '98], ɸ [SRDS '04], Falcon [SOSP '11], Pigeon [NSDI '13]
- **Monitoring and Tracing**
  - Magpie [OSDI '04], X-Trace [NSDI '07], Dapper [Google TR '10], Pivot Tracing [SOSP '15]
- **Accountability**
  - PeerReview [SOSP '07]
- **Gray Failures**
  - Fail-Stutter [HotOS '01], Limplock [SoCC '13], Fail-Slow Hardware [FAST '18], Gray Failure [HotOS '17]

Panorama's major contribution: a new way of constructing failure detectors for gray failures

Conclusion

Detect what the "requesters" see

- **Failures that matter are observable to requesters**
  - Turn **error *handlers*** into **error *reporters***
  - But must deal with design patterns that reduce observability
- **Panorama enables construction of *in-situ* observers**
  - Detects 8 crash failure and 15 gray failures in 4 real-world systems faster

https://github.com/ryanphuang/panorama

Backup Slides

Performance of Panorama Core API

Report	ReportAsync	Judge	Propagate
114.6 μs	0.36 μs	109.0 μs	776.3 μs

Local RPC

Library Call

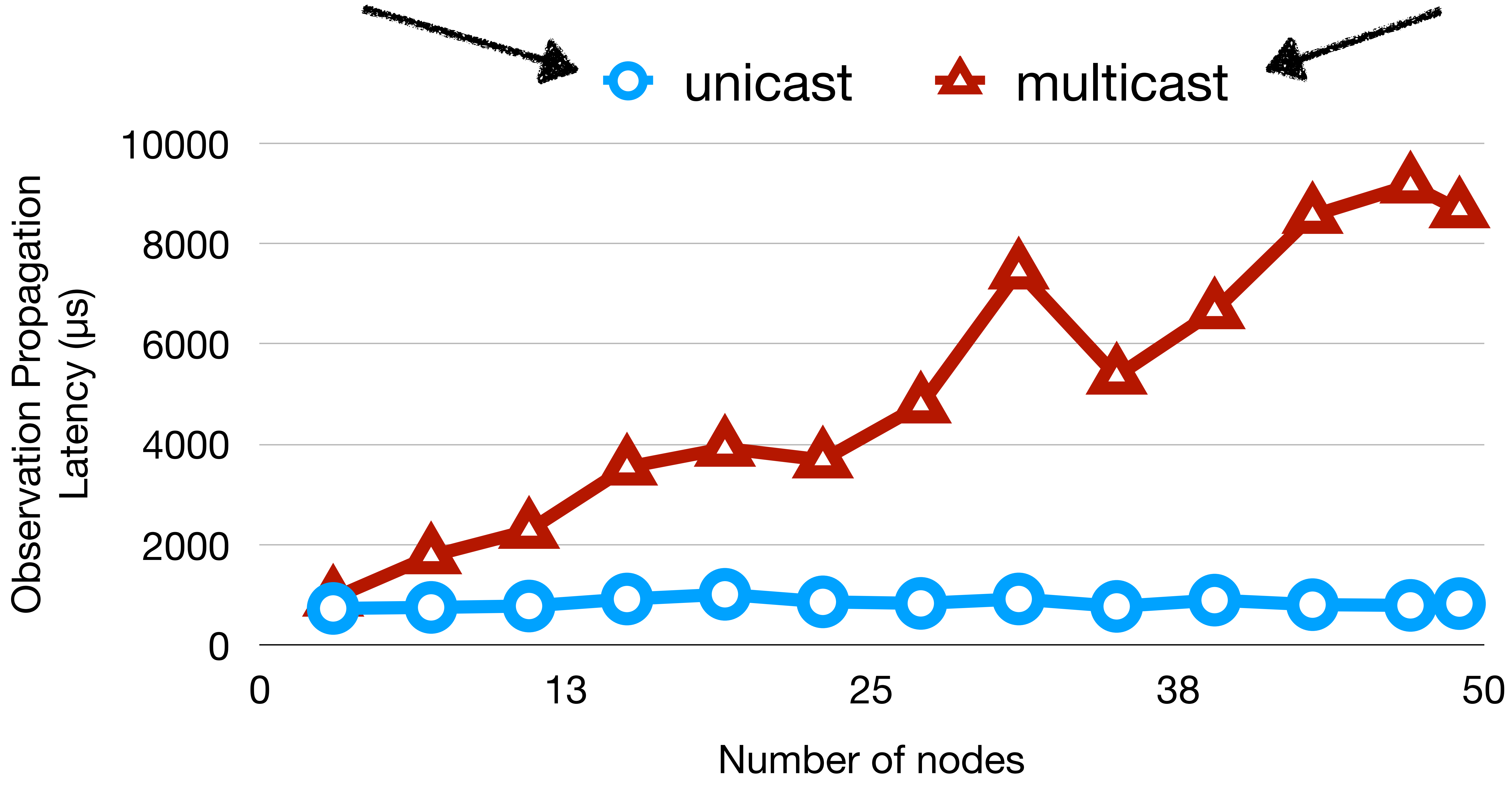
RPC

Main overhead perceived by the *in-situ* observers

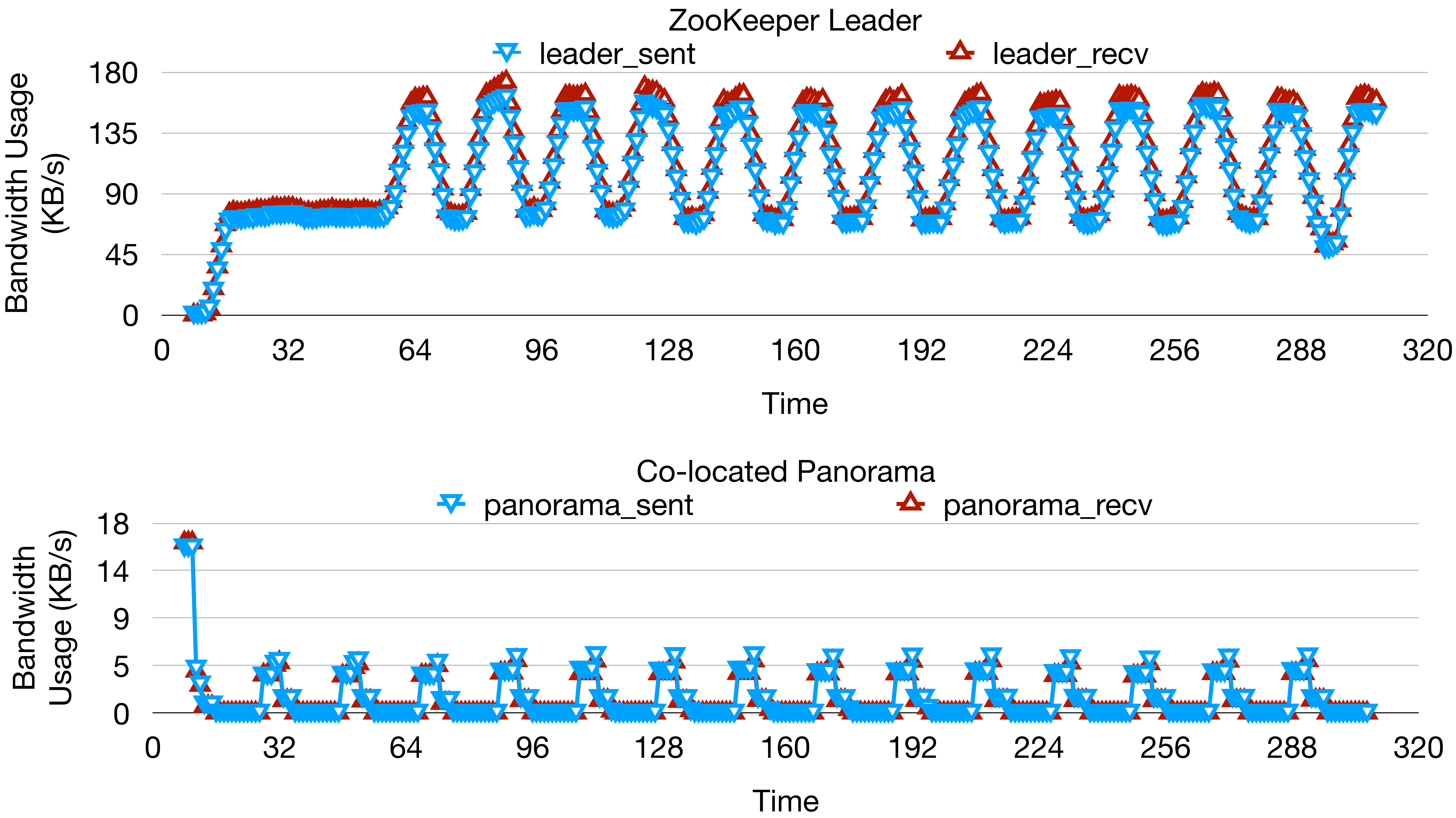
Scalability

Exchange an observation with a single Panorama instance

Exchange an observation with a clique of Panorama instance

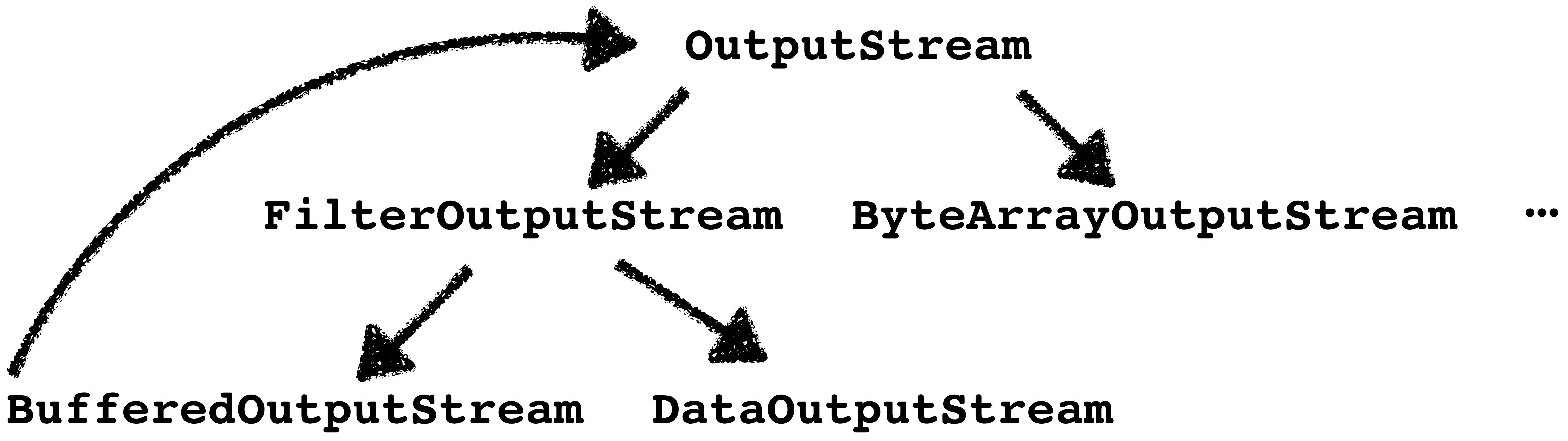


Network Bandwidth Usage

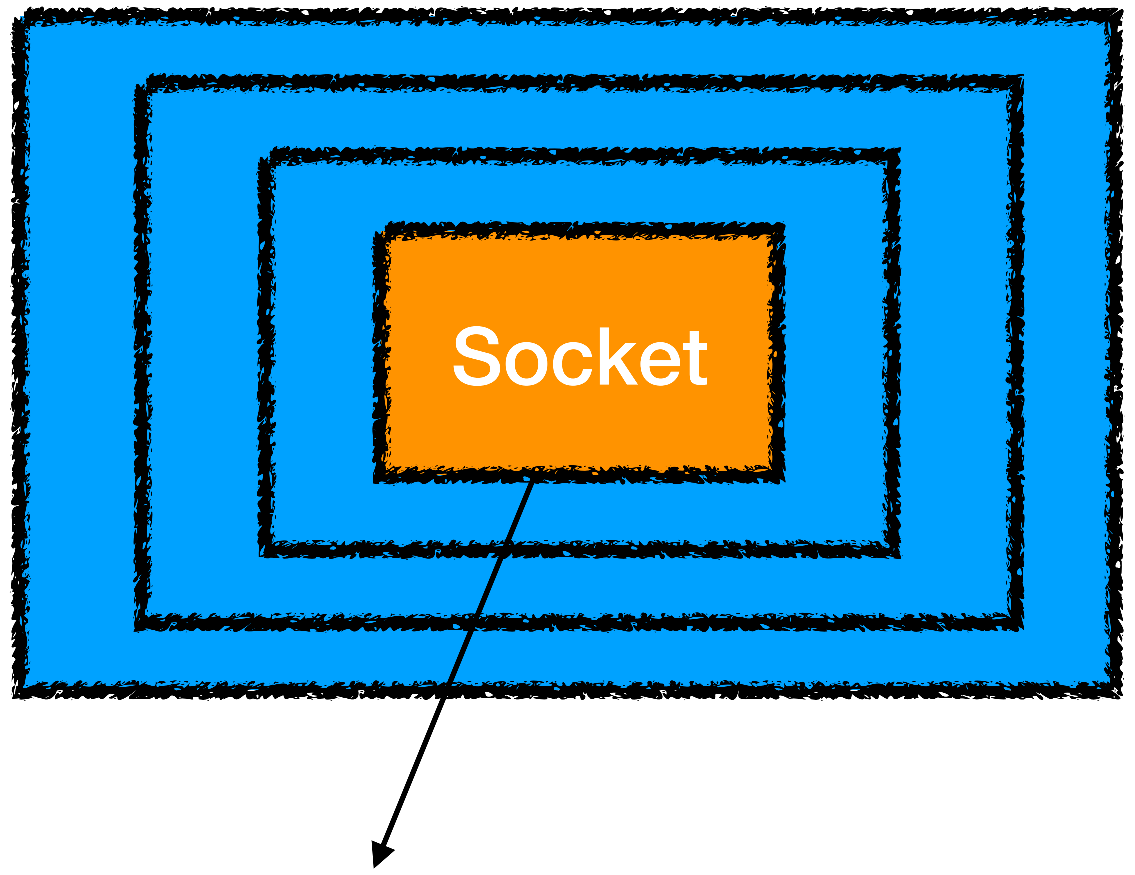


Challenge: Polymorphism and Interface

- May not distinguish ob-boundary and normal invocations
  - e.g., `void java.io.OutputStream.write(byte[] b)` could be local write or remote write




```
if (strm instanceof BufferedOutputStream) {
 ???
}
```




```
public OutputStream Socket.getOutputStream()
```

Challenge: Polymorphism and Interface

- May not distinguish ob-boundary and normal invocations
- Interface does not contain enough information (lack of fields)

```
/**
 * A BlockReader is responsible for reading a single block from a single datanode.
 */
public interface BlockReader {
  int read(byte[] buf, int off, int len) throws IOException;
  long skip(long n) throws IOException;
  int available() throws IOException;
}
```

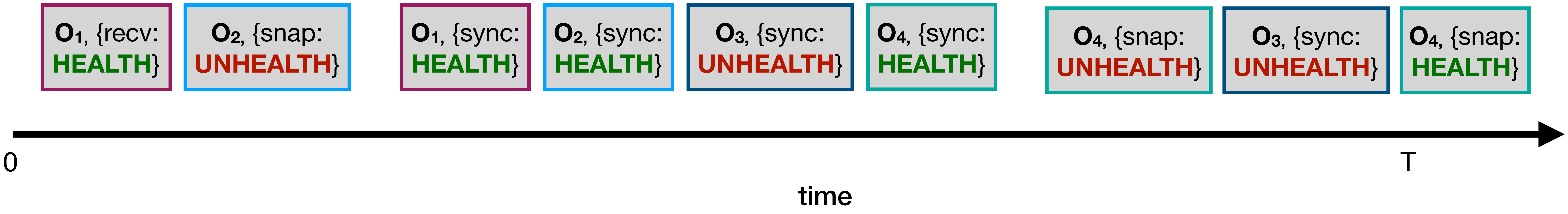
Solution

- **Change the constructors of subclasses for the abstract type**
  - return a compatible wrapper with additional subject identity field
  - set the field for remote types (e.g., `java.net.Socket.getOutputStream()`)
  - check this field at runtime to distinguish ob-boundary from normal types
- **Extend the interface with additional subject related methods**
  - call setter in the implementors (e.g., `BlockReaderFactory`)

Design Challenges

- Observations dispersed in software's source code
- Observation collection may slow down observer's normal functionality
- Diverse real-world programming paradigms affect system observability
- Observations may have biases

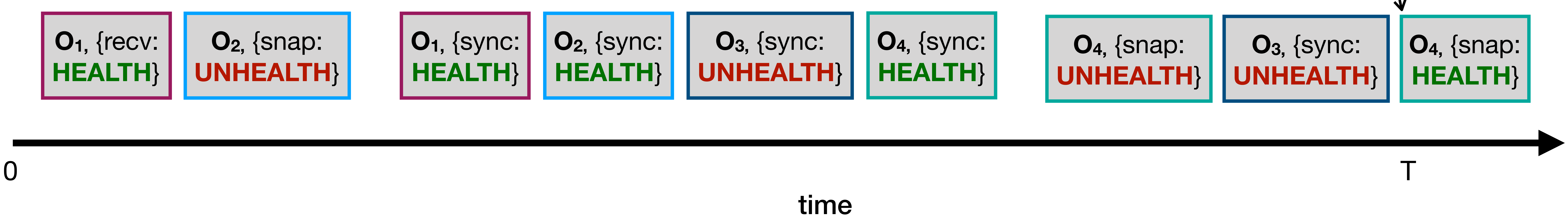
Reach Verdict from Observations



Reach Verdict from Observations

Verdict policy #1: latest observation is the verdict

- Miss intermittent failure



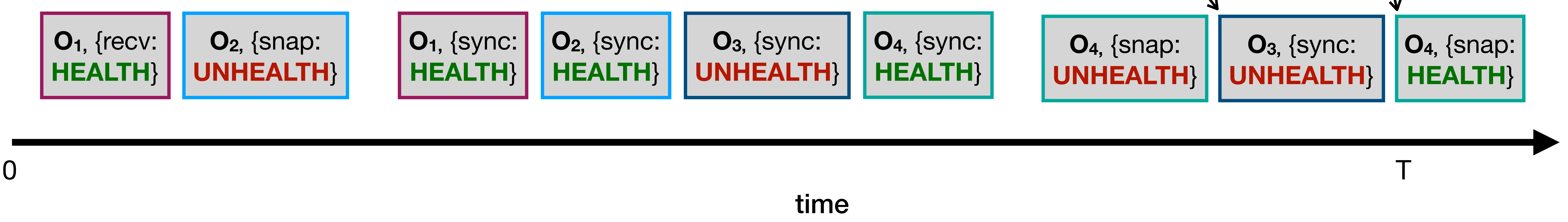
Reach Verdict from Observations

Verdict policy #1: latest observation is the verdict

- Miss intermittent failure

Verdict policy #2: unhealthy if *any* recent observation is unhealthy

- One biased observer mislead others



Reach Verdict from Observations

Verdict policy #1: latest observation is the verdict

- Miss intermittent failure

Our solution: a simple bounded look-back majority algorithm

- Group by context and observer, summarize and vote; see paper for more detail

Verdict policy #2: unhealthy if *any* recent observation is unhealthy

- One biased observer mislead others