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

Software Bugs

Misconfiguration

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

FD is a fundamental building block for fault-tolerant systems

\[
\text{Availability} \approx \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}
\]

Detecting Failure Is Crucial

FD is a fundamental building block for fault-tolerant systems

\[ \text{Availability} \approx \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \]

Improve reliability

Speed up recovery

But only if failure can be detected reliably and rapidly first

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
An Example Production Gray Failure

- Leader
- Follower
- Follower
- Follower
- ZooKeeper

Propose → Ack → Commit

Client

ZooKeeper

CREATE
WRITE
READ
CREATE
An Example Production Gray Failure
An Example Production Gray Failure

Client

ZooKeeper

Follower

Leader

Follower

Follower

Heartbeat

Heartbeat

Propose

Ack

Commit

Propose

Ack

Commit

WRITE

READ

CREATE

CREATE

CREATE
An Example Production Gray Failure

"RUOK"

Client

ZooKeeper

Leader

Follower

Follower

Follower

Follower

Heartbeat

Propose

Ack

Commit

Heartbeat

Propose

Ack

Commit

WRITE

READ

CREATE

CREATE

CREATE
An Example Production Gray Failure

![Diagram of a distributed system showing interaction between followers, leader, and ZooKeeper, with messages like "RUOK", "IMOK", propose, ack, commit, create, read, and status updates.]
Root Cause of the Gray Failure

```java
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*

Cassandra

Request Processor

Serialize

Heartbeat

Leader

Follower

Lock timeout

Lock

Unhandled requests accumulating

Socket I/O exception

GetSnapshot

Cassandra Lock timeout

Cassandra Lock
Insight 2: From Error Handling to Error Reporting

```java
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**

```java
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

```
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        LOG.error(e);
        retry();
    }
}
...  
```

```
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
...  
```

Automatically convert a software component into an *in-situ* observer
Overview of Panorama - Runtime

```java
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        + report_observation(t, e);
        LOG.error(e);
        retry(e);
    }
}
```
Overview of Panorama - Runtime

```c
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
```
Overview of Panorama - Runtime

Instrumented Software

```c
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
```

Component A

Component B

EndPoint

Panorama Instance

Observer

observability hooks

+ Component A

+ Component B

+ EndPoint

+ Observer

+ Panorama Instance

+ Instrumented Software
Overview of Panorama - Runtime

```
void func() {
    try {
        sync(t);
    }
    catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
```
Overview of Panorama - Runtime

```c
void func() {
    try {
        sync(t);
    } catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
```

Component A

Component B

EndPoint

Observer

Panorama Instance

Subject

Context

Instrumented Software

observability hooks
Overview of Panorama - Runtime

```c
    void func() {
      try {
        sync(t);
      } catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
      }
    }
```

Component A

Component B

Subject

Observer

EndPoint

Local Observation Store (LOS)

Panorama Instance

Local RPC

SubmitReport(subject, observation, context)

Context

Instrumented Software

+ observability hooks

Component A

Panorama Instance
Overview of Panorama - Runtime

```c
void func() {
  try {
    sync(t);
  } catch (RemoteError e) {
    report_observation(t, e);
    LOG.error(e);
    retry();
  }
}
```

Component A

Panorama Instance

Component B

Subject

Observer

EndPoint

Local RPC

SubmitReport(subject, observation, context)

Local Observation Store (LOS)

Propagate(report)

Context

Instrumented Software

observability hooks

+ + + + +

Component A

Panorama Instance

EndPoint

Subject

Component B

Observer

RemoteError e

LOG.error(e)

retry();

report_observation(t, e);
Overview of Panorama - Runtime

void func() {
  try {
    sync(t);
  } catch (RemoteError e) {
    report_observation(t, e);
    LOG.error(e);
    retry();
  }
}

Component A

Component B

Subject

Observer

EndPoint

Local Observation Store (LOS)

Panorama Instance

SubmitReport(subject, observation, context)

Local RPC

Learn(report)

Propagate(report)

Context

Instrumented Software

+ observability hooks
Overview of Panorama - Runtime

Instrumented Software

```plaintext
void func() {
    try {
        sync(t);
    }
    catch (RemoteError e) {
        report_observation(t, e);
        LOG.error(e);
        retry();
    }
}
```

Component A

Panorama Instance

Component B

Subject

Observer

EndPoint

SubmitReport(subject, observation, context)

Local RPC

Judge (subject)

Verdict

Local Observation Store (LOS)

Learn(report)

Propagate(report)

Context

+ observability hooks

+ ..
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 ① : locate domain-crossing invocations (*ob-boundary*)

• Step ② : identify observer and observed

• Step ③ : extract observation (*ob-point*)
Example of Observability Analysis

```java
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

```java
void deserialize(DataTree dt, InputArchive ia) {
    DataNode node = ia.readRecord("node");  // 1 locate ob-boundary
    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

1. locate ob-boundary

2. identify observer and subject

```java
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

1. locate ob-boundary

2. identify observer and subject

```java
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 Datatreex");
    }
    dt.add(node);
}

void snapshot() {
    InputArchive ia = BinaryInputArchive.getArchive(sock.getInputStream());
    try {
        deserialize(getDataTree(), ia);
    } catch (IOException e) {
        sock.close();
    }
}
Example of Observability Analysis

1. locate ob-boundary

2. identify observer and subject

3. extract observation (ob-point)  
(invalid reply)
Example of Observability Analysis

1. locate ob-boundary

2. identify observer and subject

3. extract observation (ob-point)

4. extract observation (ob-point)

```java
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();
    }
}
```
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.*()
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
Identify Observed (Subject)

- Information is in the argument or object of ob-boundary
  - `public int sendRR(MessageOut message, InetAddress to, IAsyncCallback cb)`
  - `public NNCommand NNProtocolTranslatorPB.startCheckpoint(Registration registration)`

- Sometimes the information is provided by a proxy
  - `rpc HRegionServer.RegionServerStatusService.*(*)`

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

public NNCommand NNProtocolTranslatorPB.startCheckpoint(Registration registration)
```
3 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
3 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

OutOfMemoryError due to corrupt packet
Learning from a Subtle Case

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

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

public class PrepRequestProcessor extends Thread {
    BlockingQueue<Request> requests = new ...;
    public void processRequest(Request request) throws RequestProcessorException {
        requests.add(request);
    }
}
Learning from a Subtle Case

- Indirection can *reduce* failure observability

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

Interaction of two components $C_1$ and $C_2$

$C_1$'s failure is observable to $C_2$

$C_2$'s failure is observable to $C_1$
Observability Design Patterns

C1's failure is observable to C2

C2's failure is unobservable to C1

C1's failure is observable to C2

No indirection

Interaction of two components C1 and C2
Observability Design Patterns

C2’s failure is unobservable to C1
C1’s failure is observable to C2

Interaction of two components C₁ and C₂

No indirection
Observability Design Patterns

Interaction of two components $C_1$ and $C_2$

No indirection

Request indirection
Observability Design Patterns

Interaction of two components $C_1$ and $C_2$
Observability Design Patterns

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

  ![Diagram](positive, but temporary and weak evidence) ![Diagram](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 \textbf{HEALTHY} observation merges a past \textbf{PENDING} observation
  - the positive observation is now complete

- A later \textbf{UNHEALTHY} observation overrides a past \textbf{PENDING} observation

- If \textbf{PENDING} observations outstanding for too long, degrade to \textbf{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

Integration with real-world systems

Detection Service

1. Practicality and Effort
2. Detection Time
3. Accuracy
4. Overhead

Catching crash and gray failures
Reacting to transient failures
Cost for observers
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

<table>
<thead>
<tr>
<th>ID</th>
<th>System</th>
<th>Fault Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>f₁</td>
<td>ZooKeeper</td>
<td>faulty disk in leader causes cluster lock-up</td>
</tr>
<tr>
<td>f₂</td>
<td>ZooKeeper</td>
<td>transient network partition leads to prolonged failures in serving</td>
</tr>
<tr>
<td>f₃</td>
<td>ZooKeeper</td>
<td>corrupted packet in de-serialization</td>
</tr>
<tr>
<td>f₄</td>
<td>ZooKeeper</td>
<td>transaction thread exception</td>
</tr>
<tr>
<td>f₅</td>
<td>ZooKeeper</td>
<td>leader fails to write transaction log</td>
</tr>
<tr>
<td>f₆</td>
<td>Cassandra</td>
<td>response drop blocks repair operations</td>
</tr>
<tr>
<td>f₇</td>
<td>Cassandra</td>
<td>stale data in leads to wrong node states</td>
</tr>
<tr>
<td>f₈</td>
<td>Cassandra</td>
<td>streaming silently fail on unexpected error</td>
</tr>
<tr>
<td>f₉</td>
<td>Cassandra</td>
<td>commitlog executor exit causes GC storm</td>
</tr>
<tr>
<td>f₁₀</td>
<td>HDFS</td>
<td>thread pool exhaustion in master</td>
</tr>
<tr>
<td>f₁₁</td>
<td>HDFS</td>
<td>failed pipeline creation prevents recovery</td>
</tr>
<tr>
<td>f₁₂</td>
<td>HDFS</td>
<td>short circuit reads blocked due to death of domain socket</td>
</tr>
<tr>
<td>f₁₃</td>
<td>HDFS</td>
<td>blockpool fails to initialize but continues</td>
</tr>
<tr>
<td>f₁₄</td>
<td>HBase</td>
<td>dead root drive on region server</td>
</tr>
<tr>
<td>f₁₅</td>
<td>HBase</td>
<td>replication stalls with empty WAL files</td>
</tr>
</tbody>
</table>
# Real-World Gray Failures

<table>
<thead>
<tr>
<th>ID</th>
<th>System</th>
<th>Fault Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>f₁</td>
<td>ZooKeeper</td>
<td>faulty disk in leader causes cluster lock-up</td>
</tr>
<tr>
<td>f₂</td>
<td>ZooKeeper</td>
<td>transient network partition leads to pro-longed failures in serving</td>
</tr>
<tr>
<td>f₃</td>
<td>ZooKeeper</td>
<td>corrupted packet in de-serialization</td>
</tr>
<tr>
<td>f₄</td>
<td>ZooKeeper</td>
<td>transaction thread exception</td>
</tr>
<tr>
<td>f₅</td>
<td>ZooKeeper</td>
<td>leader fails to write transaction log</td>
</tr>
<tr>
<td>f₆</td>
<td>Cassandra</td>
<td>response drop blocks repair operations</td>
</tr>
<tr>
<td>f₇</td>
<td>Cassandra</td>
<td>stale data in leads to wrong node states</td>
</tr>
<tr>
<td>f₈</td>
<td>Cassandra</td>
<td>streaming silently fail on unexpected error</td>
</tr>
<tr>
<td>f₉</td>
<td>Cassandra</td>
<td>commitlog executor exit causes GC storm</td>
</tr>
<tr>
<td>f₁₀</td>
<td>HDFS</td>
<td>thread pool exhaustion in master</td>
</tr>
<tr>
<td>f₁₁</td>
<td>HDFS</td>
<td>failed pipeline creation prevents recovery</td>
</tr>
<tr>
<td>f₁₂</td>
<td>HDFS</td>
<td>short circuit reads blocked due to death of domain socket</td>
</tr>
<tr>
<td>f₁₃</td>
<td>HDFS</td>
<td>blockpool fails to initialize but continues</td>
</tr>
<tr>
<td>f₁₄</td>
<td>HBase</td>
<td>dead root drive on region server</td>
</tr>
<tr>
<td>f₁₅</td>
<td>HBase</td>
<td>replication stalls with empty WAL files</td>
</tr>
</tbody>
</table>

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 $f_1$

- **Failure reports**
- **Client Latency (ms)**
- **Client Error**
- **Timeout**
- **Success**
- **Panorama Observer**
- **Fault occurs**
- **Fault clears**

Client executing mixed workloads
Transient Failures

- context1: all
- context3: RecvWorker
- context5: start
- context7: QuorumCtxnManager

- context2: Learner
- context4: QuorumPacket
- context6: FinalRequestProcessor

Healthy

Unhealthy
Latency Overhead to Observers

System instrumented with Panorama reporting hooks

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], ϕ [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

<table>
<thead>
<tr>
<th></th>
<th>Local RPC</th>
<th>Library Call</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>114.6 μs</td>
<td>0.36 μs</td>
<td></td>
</tr>
<tr>
<td>ReportAsync</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Judge</td>
<td>109.0 μs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Propagate</td>
<td>776.3 μs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Observation Propagation Latency (μs)

Number of nodes

unicast
multicast
Network Bandwidth Usage

ZooKeeper Leader
- leader_sent
- leader_recv

Co-located Panorama
- panorama_sent
- panorama_recv
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

```java
if (strm instanceof BufferedReader) {
    ???
}
```

```java
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)

```java
/**
 * 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

O₁, {recv: HEALTH} O₂, {snap: UNHEALTH} O₁, {sync: HEALTH} O₂, {sync: HEALTH} O₃, {sync: UNHEALTH} O₄, {sync: HEALTH} O₄, {snap: UNHEALTH} O₃, {sync: UNHEALTH} O₄, {snap: HEALTH}

0 time T
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

**Our solution:** a simple bounded look-back majority algorithm

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

**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

---

- $O_1$, {recv: HEALTH}
- $O_2$, {snap: UNHEALTH}
- $O_1$, {sync: HEALTH}
- $O_2$, {sync: HEALTH}
- $O_3$, {sync: UNHEALTH}
- $O_4$, {sync: HEALTH}
- $O_4$, {snap: UNHEALTH}
- $O_3$, {sync: UNHEALTH}
- $O_4$, {snap: HEALTH}