Uncovering Distributed Storage System Bugs in Testing (not in Production!)

Cheng Huang, Shaz Qadeer

09.19.2016

Top Problem in Cloud Storage – Testing Coverage

- "Due to limited testing coverage, many correctness problems are only exposed in production through live-sites"
- "Engineering overhead extremely high to identify problems"
- "Practical tools that can improve testing coverage highly appreciated!"

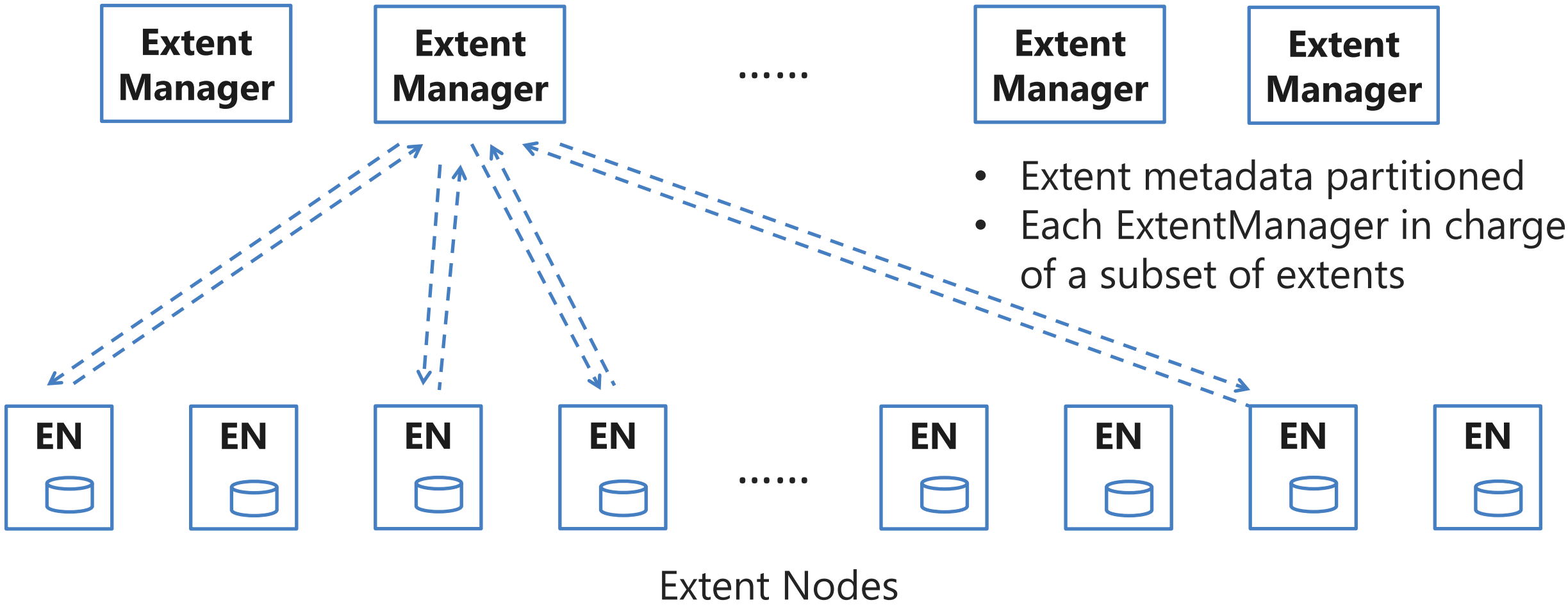
– technical leaders and senior managers in Azure Storage

Azure Storage vNext

- Azure Storage
  - 10s PB in 2010 → EB in 2015
  - 60+ trillion objects
  - Paxos-based, centralized metadata management

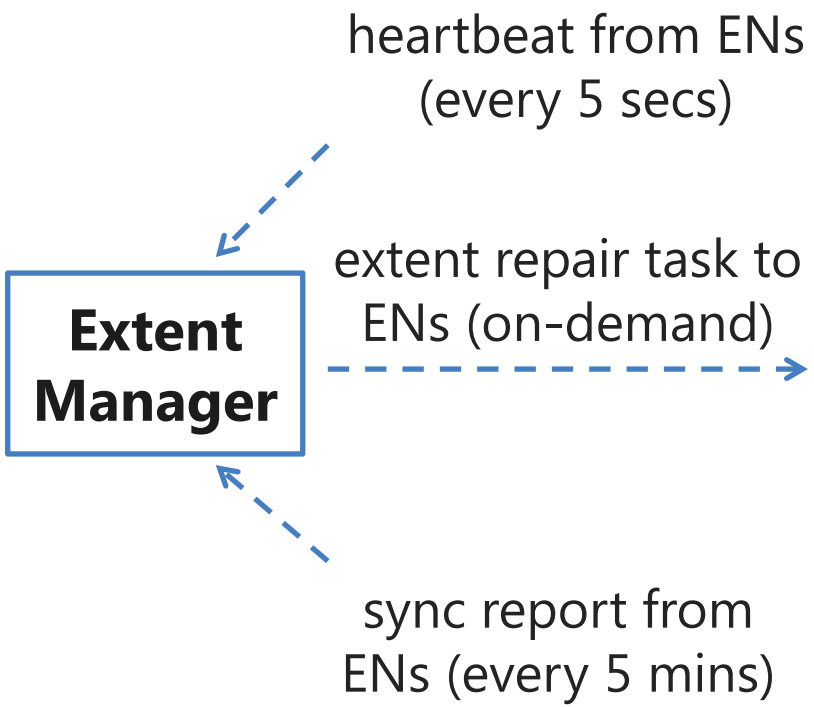
- vNext: new architecture to scale capacity by more than 100x
  - Completely distributed and fully scale-out metadata management
  - Data stored in extents (GB per extent) and streams (list of extents)
  - Extents and streams managed by distributed, light-weight Extent Managers and Stream Managers

Extent Management in Azure Storage vNext



- Extent metadata partitioned
- Each ExtentManager in charge of a subset of extents

Replication Logic in Extent Manager



- Extent Manager maintains 3 replicas for every extent
  - Discover node failures (heartbeat)
  - Identify missing replica
    - Sync report lists all extents on EN
  - Schedule extent repair task

Difficulty in Testing vNext

- Unit tests
  - Emulate heartbeat, sync report, EN expiration
  - Verify Extent Manager behavior

- Integration tests
  - Launch real Extent Manager and Extent Nodes
  - Kill EN and launch new EN → verify extents repaired
- Unit tests & integration tests always pass

- Stress tests fail from time to time, when repair gets stuck as
  - Many extents are created
  - ENs are constantly killed and launched

Result Preview

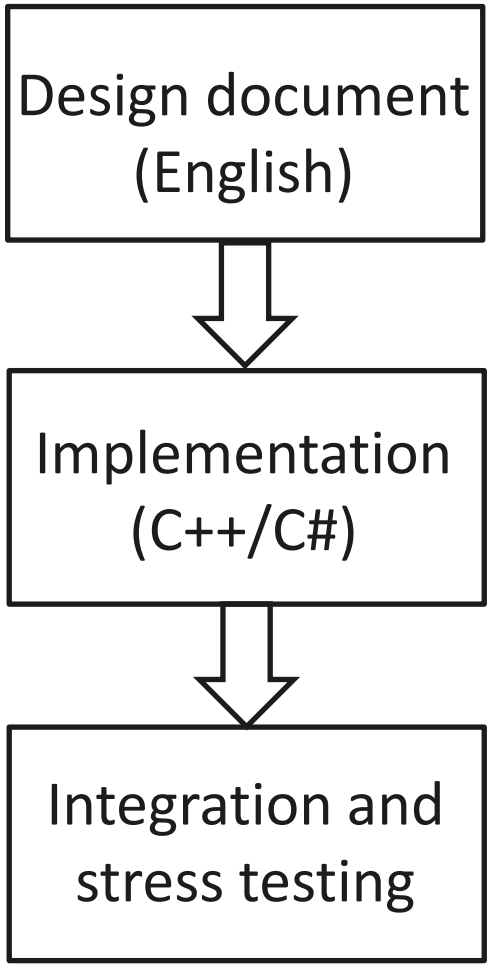
- You are going to see some magic next …

- What did we get out of this?
  - Fast re-pro → less than minute before re-producing the issue
  - Small trace → bug confirmed by examining the trace
  - If cannot identify culprit
    - Iterate and refine debug outputs (fast turn around helps)
  - Implemented and verified the fix → less than one hour

Characteristics of distributed storage systems

- Concurrency
  - multiple processes communicating with each other
- Failures:
  - Messages may be dropped
  - Compute nodes may fail
- Extensive use of timers
  - To (approximately) detect failures

Current practice of distributed storage systems



Informal description
(English prose, State-machine diagrams)
leaves many corner cases unexplored

Inadequate support for specification and testing of concurrent asynchronous code

Inadequate test coverage
Difficult to debug problems

TLA+/PlusCAL

- Temporal logic of actions
  - Model checker TLC for small protocol configurations
  - Used in Amazon, Microsoft (XStore, Cosmos, MSR), Intel, and academia
- Specify protocol using mathematical formulas
  - with sugar for control flow and concurrency
  - safety and liveness properties
- References
  - Who Builds a Skyscraper Without Drawing Blueprints? (Leslie Lamport, presentation video available on http://resnet, March 2014)
  - How Amazon Web Services Uses Formal Methods (CACM, April 2015)

But …

- TLA+ is great for modeling protocol specifications
  - But what about protocol implementations?

The P programming language

- State machines (actors) communicating via message passing
  - Systematic testing similar to TLC
  - Compiles to executable C code

- Experience inside Microsoft
  - Windows 8 USB 3.0 & Windows Phone USB
    - Found and fixed hundreds of bugs
  - Hololens firmware driver stack
    - “After a year of heavy daily usage across the entire team there have been no bugs in the layers that were modeled and verified.”

But …

- TLA+ is great for modeling protocol specifications
  - But what about protocol implementations?

- P is great for validating protocol specifications and writing new protocol implementations
  - But what about testing existing implementations?

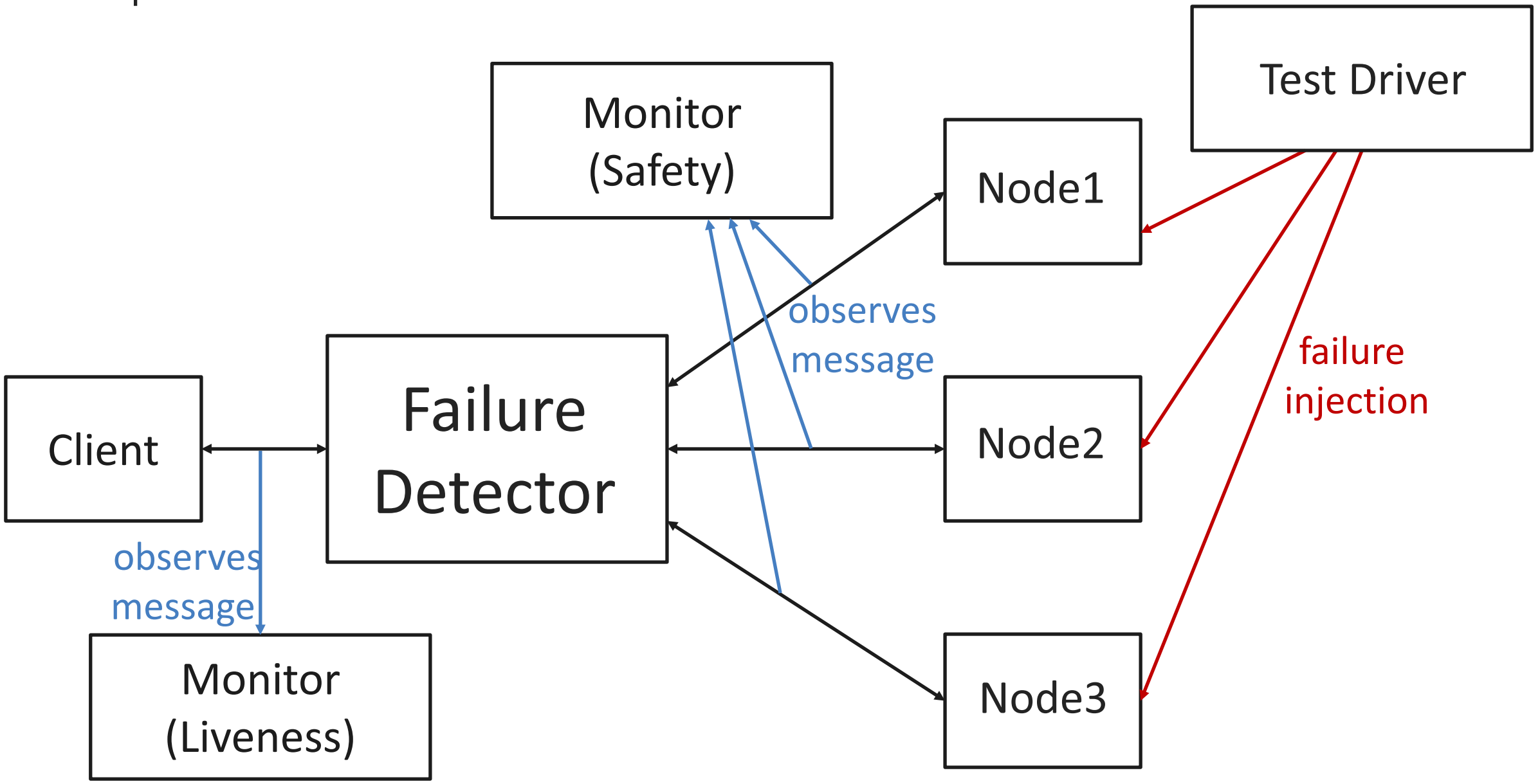
Specification
- Safety
- Liveness

Modeling
- Concurrency
- Failures

Testing
- Harnesses
- Mocking

P# language and test framework
- State machines (actors) in C#
- Extension of C#
- Primitives for modeling and specification
- Runtime controls and explores nondeterminism

Example: Failure Detector



Specifying distributed systems

- Safety properties
  - Generalizes assertions
  - Something bad does not happen
  - Violation exhibited by a finite execution

- Liveness properties
  - Generalizes termination
  - Something good eventually happens
  - Violation exhibited by an infinite execution

Safety

```
monitor Safety {
   int pending;

   [OnEvent(PING, ProcessPing)]
   void ProcessPing ( ) {
      pending = pending + 1;
      assert (pending <= 3);
   }

   [OnEvent(PONG, ProcessPong)]
   void ProcessPong ( ) {
      assert (0 < pending);
      pending = pending - 1;
   }
}
```

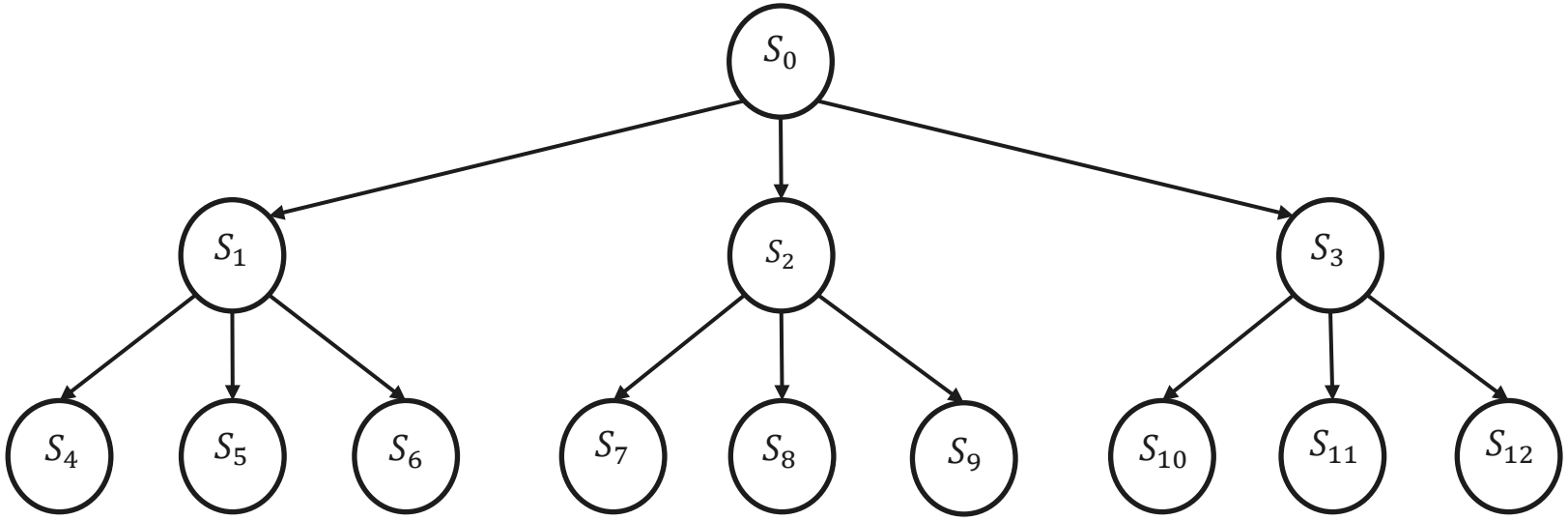
- Messages between Failure Detector and all nodes are observed by the safety monitor
- Safety monitor checks safety property: *Assert* (# of pending PING w/o PONG <= 3);

Liveness

```
monitor Liveness {
   HashSet<Node> alive;

   hot state ShuttingDown {

      [OnEvent(DOWN, OnNodeDown)]
      void OnNodeDown(Node n) {
         alive.remove(n);
         if (alive.Empty()) jumpto Finished;
      }

   }

   cold state Finished { }
}
```

- Messages between Failure Detector and client are observed by the liveness monitor
- Liveness monitor checks liveness property: *Assert* (temperature < ∞);

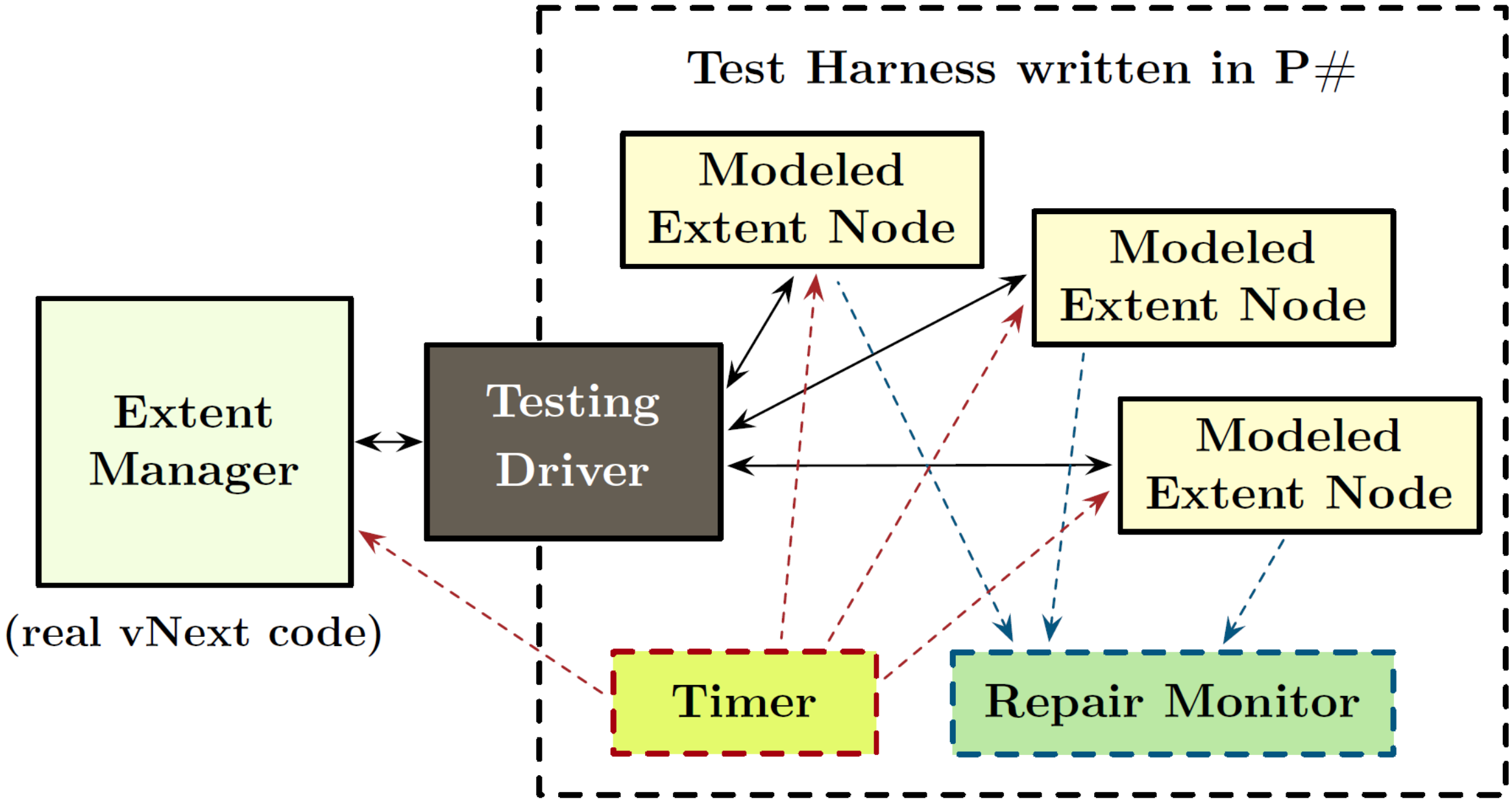
Systematic testing as a search problem



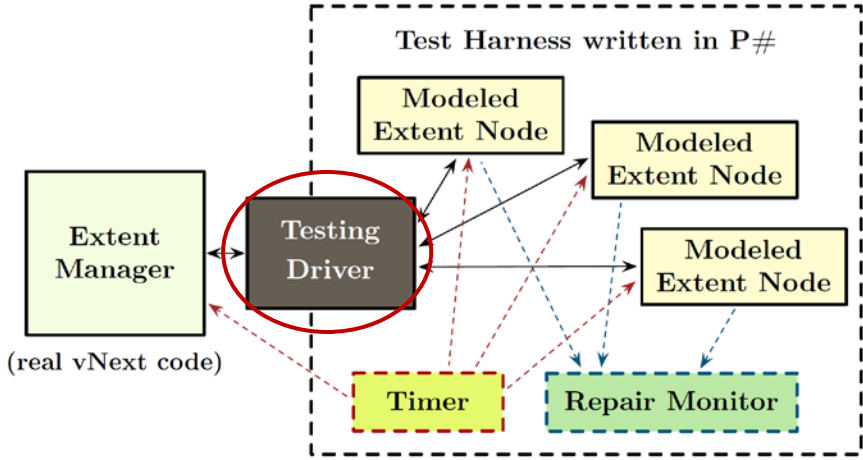
Search strategies

- Exhaustive search
  - depth-first
  - breadth-first
- Randomized search
  - Random walk
- Prioritized search
  - Custom schedulers for directing search
  - Exhaustive or randomized search

Testing Extent Manager with P#



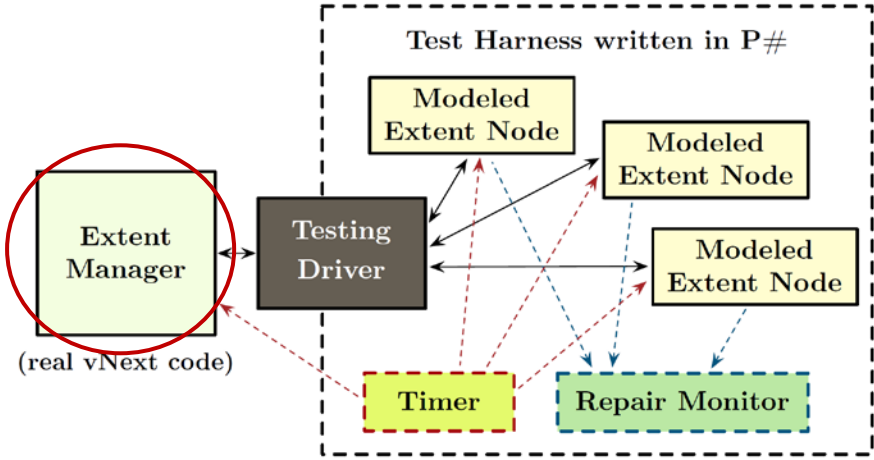
Testing Driver



- Setting up the "distributed" system
  - 1 Extent Manager, 3 ENs and single $extent_{one}$
  - Small setup sufficient to expose bug → easy to debug
- Two testing scenarios
  - Scenario I: drop single $extent_{one}$ to one EN
    - Assert ($extent_{one}$ *eventually* replicated to the other ENs)
  - Scenario II: fail arbitrary EN and launch a new one
    - Assert ($extent_{one}$ *eventually* replicated to the new EN)
- Non-determinism given to P#
  - EN: heartbeat, sync report, failure
  - Extent Manager: schedule repair task
  - Message: delay and loss

Extent Manager P# Machine

```
// wrapping the target vNext component in a P# machine
class ExtentManagerMachine : Machine {
  private ExtentManager ExtMgr; // real vNext code

  void Init() {
    ExtMgr = new ExtentManager();
    ExtMgr.NetEngine = new MockedNetEngine(); // mock network
    ExtMgr.IsMockingTimer = true; // disable internal timer
  }

  [OnEvent(ExtentNodeMessageEvent, DeliverMessage)]
  void DeliverMessage(ExtentNodeMessage msg) {
    // relay messages from Extent Node to Extent Manager
    ExtMgr.ProcessMessage(msg);
  }
```

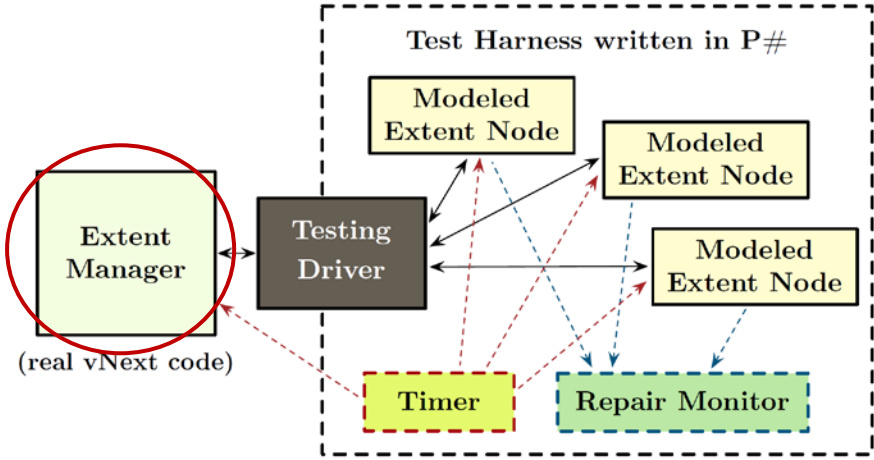


wrap testing target

- instantiate target
- mocked network for outbound messages

relay inbound messages from ENs

Outbound Messages



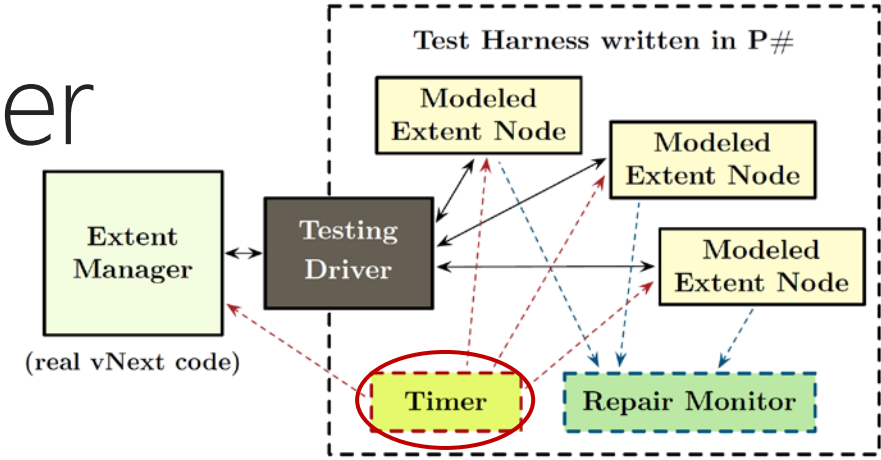

```
// network interface in vNext
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}
```

real network engine

```
// mocked engine for intercepting Extent Manager messages
class MockedNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // intercept and relay Extent Manager messages
    PSharpRuntime.Send(this.TestingDriver,
      new MessageFromExtentManagerEvent(), s, msg);
  }
}
```

mocked network:
intercept and relay
outbound messages to P#

Extent Manager Driven by P# Timer

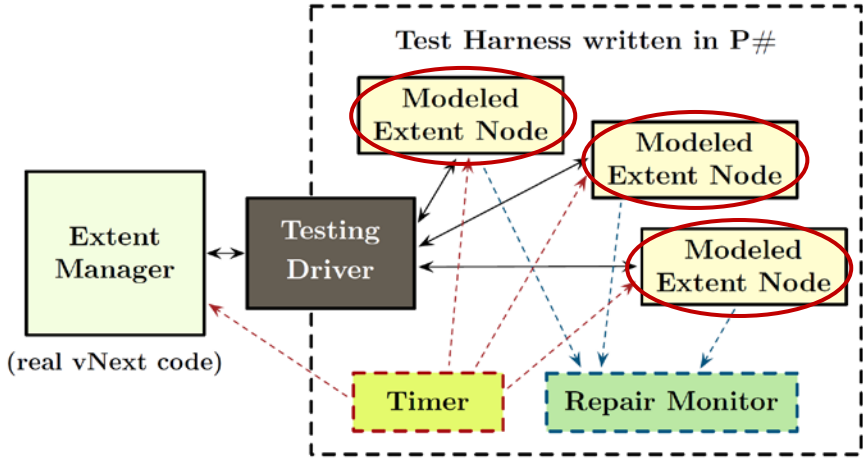



```
// wrapping the target vNext component in a P# machine
class ExtentManagerMachine : Machine {
  private ExtentManager ExtMgr; // real vNext code

  void Init() {
    ExtMgr = new ExtentManager();
    ExtMgr.NetEngine = new MockedNetEngine(); // mock network
    ExtMgr.IsMockingTimer = true; // disable internal timer
  }

  [OnEvent(ExtentNodeMessageEvent, DeliverMessage)]
  void DeliverMessage(ExtentNodeMessage msg) {
    // relay messages from Extent Node to Extent Manager
    ExtMgr.ProcessMessage(msg);
  }

  [OnEvent(TimerTickEvent, ProcessExtentRepair)]
  void ProcessExtentRepair() {
    // extent repair loop driven by external timer
    ExtMgr.ProcessEvent(new ExtentRepairEvent());
  }
}
```

disable internal timer

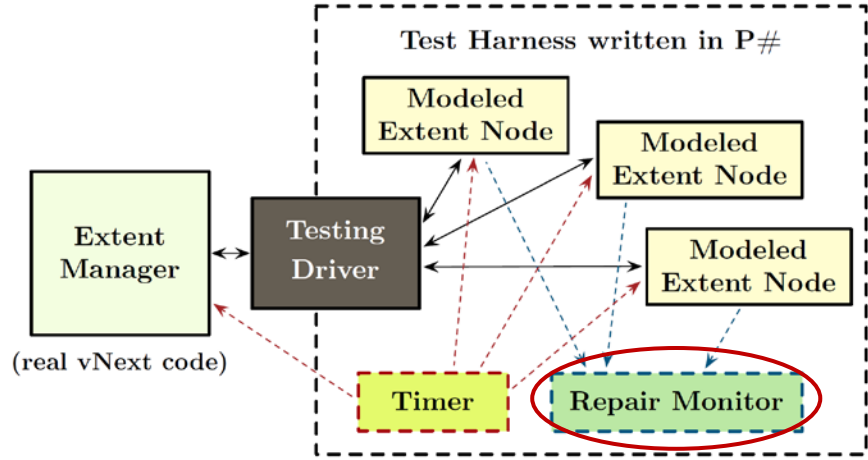
act upon P# timer

Mocked Extent Node



- Option I
  - Wrap real EN inside a P# machine
- Option II (what we did in this case)
  - Mock simplified EN logic only related to replication
    - Heartbeat, sync report and extent repair
  - Re-use EN internal components whenever appropriate

Repair Monitor



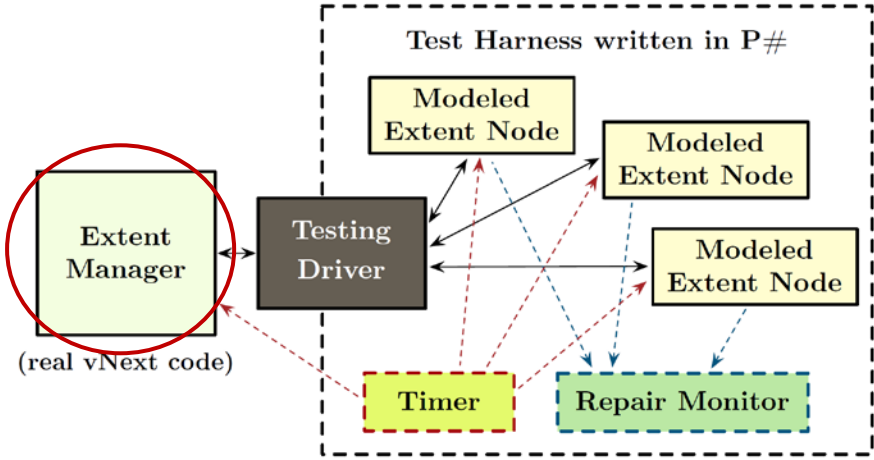

```
class RepairMonitor : Monitor {
  private HashSet<Machine> ExtentNodesWithReplica;

  // cold state: repaired
  cold state Repaired {
    [OnEvent(ENFailedEvent, ProcessENFailure)]
    void ProcessENFailure(ExtentNodeMachine en) {
        ExtentNodesWithReplica.Remove(en);
        if (ReplicaCount < Harness.REPLICA_COUNT_TARGET)
          jumpto Repairing;
    }
  }

  // hot state: repairing
  hot state Repairing {
    [OnEvent(ExtentRepairedEvent, ProcessRepairCompletion)]
    void ProcessRepairCompletion(ExtentNodeMachine en) {
      ExtentNodesWithReplica.Add(en);
      if (ReplicaCount == Harness.REPLICA_COUNT_TARGET)
        jumpto Repaired;
    }
  }
}
```

cold state: repaired

hot state: repairing

stuck in hot state infinitely long → liveness bug

Liveness Bug in Extent Manager



sync report

heartbeat

Extent Manager internals

Extent Center
(extent → EN locations)

update ExtCtr upon sync

EN Map (EN → last heartbeat)

refresh EN Map upon heartbeat

EN expiration loop

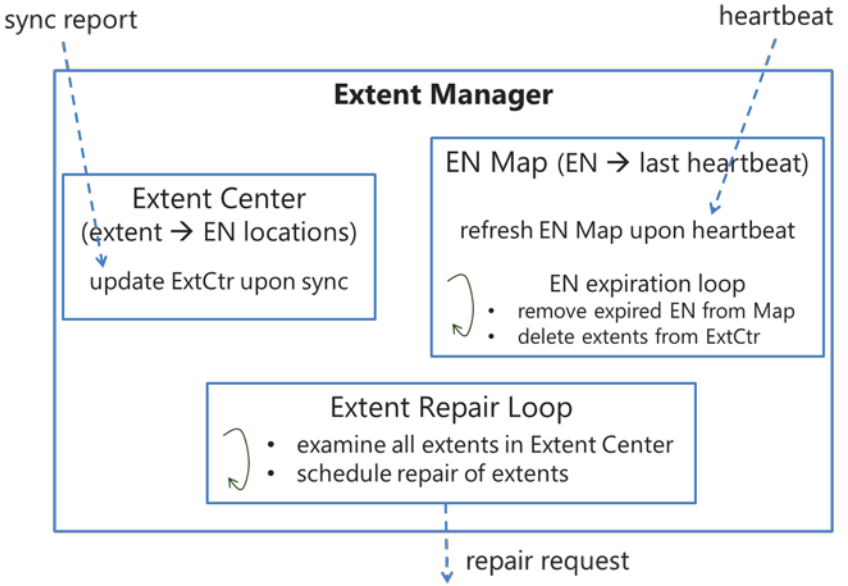
- remove expired EN from Map
- delete extents from ExtCtr

Extent Repair Loop

- examine all extents in Extent Center
- schedule repair of extents

repair request

Liveness Bug in Extent Manager



- EN_0 failed
  - remove EN_0 from EN Map
  - delete EN_0's extent from Extent Center
    - $(extent_{one}, (EN_0, EN_1, EN_2)) \rightarrow (extext_{one}, (EN_1, EN_2))$
- Received sync report from EN_0
  - update Extent Center
    - $(extent_{one}, (EN_1, EN_2)) \rightarrow (extent_{one}, (EN_0, EN_1, EN_2))$
- Extent repair loop never schedules repair task
    - $(extent_{one}, (EN_0, EN_1, EN_2)) \rightarrow$ all replicas are healthy
- EN_0 never deleted again
  - EN_0 no longer in EN Map

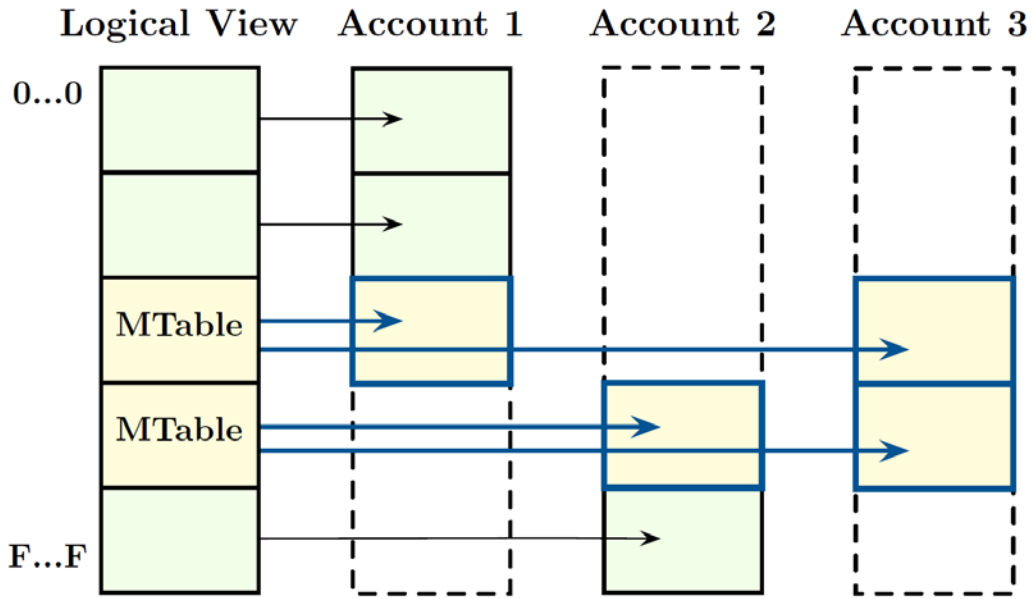
- The one liner fix → refresh EN Map upon sync report

Experience Recap

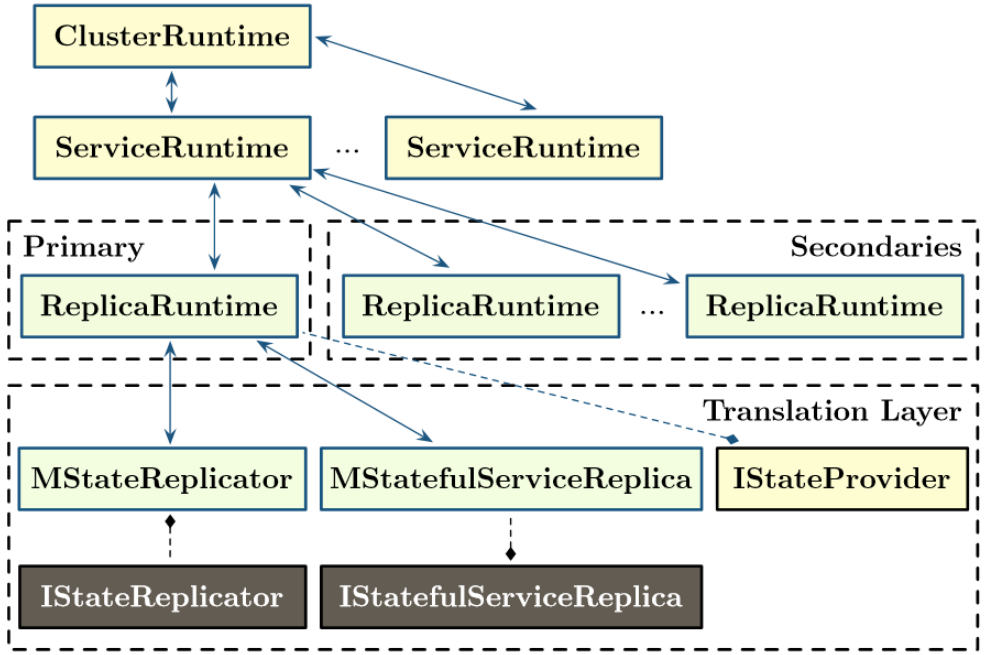
- Less than a minute before hitting the liveness bug
- Bug confirmed by examining trace
  - Replica missing, but no repair task
- Cannot identify culprit
  - Missing debug outputs
- Iterate and refine debug outputs (fast turn around helps)
  - Bug identified, fix obvious
- Implemented and verified the fix
  - 100,000 iterations, no bug → less than 1 hour

Successes in Other Teams

- Artifacts Services team
  - Azure Table live migration
  - Logical table composed of multiple physical Azure tables



- Azure Service Fabric team
  - Primary-secondary replication



numerous subtle safety bugs uncovered

Resources

- P/P# is open source at https://github.com/p-org

- A companion technical paper available (drop us a note)
  - qadeer@microsoft.com
  - Cheng.Huang@microsoft.com