Uncovering Bugs in Distributed Storage Systems During Testing (not in Production!)

Pantazis Deligiannis
Imperial College London
Top Problem in Distributed Storage Systems: **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
But why programming and testing distributed systems is so **HARD**?
Many sources of nondeterminism cause subtle (but serious) bugs that are hard to detect, diagnose and fix.
Today, to find these bugs, engineering teams use:

- Design reviews
- Code reviews
- Unit testing
- Integration testing
- Stress testing
- ...
Case Study in Microsoft: Testing Azure Storage vNext
Microsoft Azure Storage

Durable, highly available, massively scalable cloud storage solution

10s PB in 2010 \rightarrow now EB

60+ trillion objects

Paxos-based, centralized metadata management
New architecture to **scale Azure Storage capacity** by >100x

- Completely distributed and fully scale-out metadata management system
- Data stored in extents (GB per extent) — extent space partitioned
- Extent Nodes are managed by light-weight, distributed Extent Managers
Microsoft Azure Storage vNext

- One of the key tasks of Extent Manager is to maintain the replicas
- In this case study we focus on testing the replication logic — very important as we do not want to lose customer data!
Difficulty in Testing

- Unit tests — *always pass*

- Integration tests — *always pass*
  - Launch Extent Manager and Extent Nodes
  - Kill EN and launch new EN → test extents repaired

- Stress tests — *fail from time to time*
  - ENs are constantly killed and launched
  - replication process gets stuck
  - hard to figure out why — too many logs accumulated!
Testing vNext with P#

- P# [PLDI’15] is a systematic testing framework
- Controls and systematically explores all declared sources of nondeterminism in a distributed system
- Support for modeling system components as communicating state-machines to perform component-wise testing (which can scale better than testing unmodified systems)
  - Provides a send primitive for sending messages between P# machines instead of real network, and can systematically explores interleavings
  - Write test harness that injects failures, timeouts, client requests, etc
  - Write safety and liveness specifications
- Can be applied on message passing systems written in .NET or C++
- Open source in GitHub, available for anyone to use!
Bug Finding as a Search Problem

program start

BUG!
P# test harness for vNext

Wrapped in a P# state-machine

Test Harness written in P#

- Modeled Extent Node
- Modeled Extent Node
- Modeled Extent Node

Extended Manager

Testing Driver

Timer

Repair Monitor

(real vNext code)

684 lines of source code
Testing Driver

- Setting up the “distributed” system
  - P# simulates system in a single process!
  - Messages go through P#, not the real network!
  - 1 real Extent Manager, 3 modeled ENs and a single extent
  - Small setup sufficient to expose bug → easy to troubleshoot

- Non-determinism modeled in P#
  - E.g. EN failures, timeouts, etc
  - Messages: delays and losses

- Two testing scenarios
  - Scenario I: pass single extent to one EN — assert (extent eventually replicated to the other ENs)
  - Scenario II: fail arbitrary EN and launch a new one — assert (extent eventually replicated to the new EN, target is 3 replicas available)
Real Extent Manager Wrapper Machine

```csharp
// 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
        ExtMgrProcessEvent(new ExtentRepairEvent());
    }
}
```

wrap testing target (real Extent Manager)

instantiate testing target and create mock network for outbound messages

relay inbound messages from ENs to the real Extent Manager
Outbound Messages

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

// 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);
    }
}
Real Extent Manager Driven by P# Timer

```csharp
// 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());
    }
}
```
Modeled EN Components

- Simplified EN logic only related to the replication process
- Helps to achieve better testing scalability by not having to go through the real ENs
- Reuses EN internal components whenever appropriate (to maximize code reuse)
Liveness Monitor

```csharp
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)
                jump to Repairing;
        }
    }

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

Stuck in hot state infinitely long → liveness bug

cold state: liveness property satisfied

hot state: liveness property not satisfied yet
Liveness Checking in P#

- **Approach I — similar to MaceMC [NSDI’07]**
  - Run until a given large bound
  - Check liveness monitor when bound is reached
  - If in hot state, report *potential* liveness bug

- **Approach II (work-in-progress)**
  - Try to detect a *fair, infinite loop* (lasso-based approach)
  - If the monitor is *stuck in a hot state* in the loop (i.e. never goes to a cold state), we report a liveness bug
Testing vNext with P#

- Developers spent 2 weeks modeling the environment of the Extent Manager and writing the liveness specification P# monitor (684 loc)
- P# found a liveness violation in a matter of minutes and produced a small sequential trace
- Identify and fix bug by developers in less than an hour (one line of code — see next slide)
- After the fix, developers run the P# test harness for 1 hour without finding any bugs
Extent Manager

Sync report from ENs
Heartbeat from ENs
Repair requests to ENs

Extent Manager
Extent Center
Extent Node Map
Extent Repair Loop
Liveness Bug

- Extent Node EN_0 failed (from 3 available)
  - EN_0 removed from ExtentNodeMap
  - Deleted EN_0’s extent from ExtentCenter
    \[(\text{extent} \{ \text{EN}_0, \text{EN}_1, \text{EN}_2 \}) \Rightarrow (\text{extent} \{ \text{EN}_1, \text{EN}_2 \})\]
- Extent Manager received delayed sync report from EN_0
  - Updated ExtentCenter
    \[(\text{extent} \{ \text{EN}_1, \text{EN}_2 \}) \Rightarrow (\text{extent} \{ \text{EN}_0, \text{EN}_1, \text{EN}_2 \})\]
- EN_0 no longer in ExtentNodeMap \Rightarrow never deleted again from ExtentCenter
- Extent Manager never schedules repair process again
  \[(\text{extent} \{ \text{EN}_1, \text{EN}_2 \}) \Rightarrow (\text{extent} \{ \text{EN}_0, \text{EN}_1, \text{EN}_2 \}) \Rightarrow \text{all healthy!}\]
- If this happens two more times \Rightarrow all replicas lost \Rightarrow customer data lost!
- One line fix: refresh ExtentNodeMap upon sync report!
Other case studies in Microsoft

- Tools for Software Engineers (TSE) team: used P# during development of a Live Table Migration protocol for Azure (found and fixed >10 safety bugs)

- Team in MSR India: created P# executable model of Azure Service Fabric runtime, which can be eventually used to test arbitrary customer services built on top of the Service Fabric APIs
P# has been successfully used by Microsoft Azure to test multiple distributed systems.

P# is freely available in GitHub so you can use it for your own projects!

https://github.com/p-org/PSharp

p.deligiannis@imperial.ac.uk